Mahout IN ACTION

Sean Owen
Robin Anil
Ted Dunning
Ellen Friedman



MANNING

Requires Adobe Acrobat Reader to play audio and video links

Mahout in Action

Mahout in Action

SEAN OWEN
ROBIN ANIL
TED DUNNING
ELLEN FRIEDMAN



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Katharine Osborne
Copyeditor: Andy Carroll
Proofreader: Melody Dolab
Typesetter: Dottie Marsico
Cover designer: Marija Tudor

ISBN 9781935182689
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 16 15 14 13 12 11

brief contents

contents

preface

The path to here, for me (Sean), began in 2005. A friend was starting a company that would lean heavily on collaborative filtering. There were mature, open source packages for this purpose at the time, but they seemed in some ways too elaborate for simple use cases, and in other ways they seemed built for research purposes. For better or worse, I instead prototyped a simple recommender for my friend's startup, from scratch. The startup, unfortunately, cancelled itself. Nevertheless, I couldn't bring myself to delete the prototype. It was certainly interesting, so I cleaned and documented it and released it as an open source project called Taste.

Nothing happened for a year. In my spare time, I added pieces and fixed problems, and then a user or two popped up with bugs and patches—and a few more, and then several more. By 2008, there was a small but unmistakable user base out there. And the Apache Lucene folks who had just spun off machine-learning-related efforts into Apache Mahout suggested we merge. This book project began in late 2009. I find myself surprised and pleased to still be rolling along with this growing snowball of a project in 2011 as it's beginning to be used by large companies in production.

So, I'm only accidentally here. While I have been a senior engineer, formerly at Google, nobody would mistake me for a expert researcher in the field. I am more like a museum curator than a painter—collecting, organizing, and packaging for wider use the great ideas of a field. It turns out that's useful work too.

Someone recently described the book, after reading a draft, as a "pop" machine learning book. It was meant as a compliment, and I couldn't agree more. Machine learning is a bit of magic, though much of the research-oriented writing on the subject can look like arcane spells to anyone but the specialist, and can seem divorced

from the reality of applying the techniques. *Mahout in Action* aims to be accessible, to unearth the interesting nuggets of insight for the enthusiast, and to save the practitioner time in getting work done. I hope it provides you more "a-ha!" moments than "wha…?" moments.

SEAN OWEN

My (Robin's) interest in machine learning started during my days in college, back in 2006. At that time, I was working as an intern with a group of people designing a personalized recommendation engine. That group flourished and became a company called Minekey; I was invited to join as one of its core developers. The next four years of my life were spent implementing and experimenting with machine learning techniques. Somewhere along that path, I stumbled across Mahout and started contributing as a Google Summer of Code student. The next thing I knew, I was contributing algorithms and patches to its codebase, tuning and optimizing performance, and helping other folks on the mailing list.

I am really fortunate to be part of a wonderful and growing community of developers, researchers, and enthusiasts of machine learning. As more and more companies are adopting Mahout, it is becoming a mainstream library of machine learning. I really hope you enjoy reading this book.

ROBIN ANIL

I (Ted) came to the application side of projects from research in machine learning. Formerly an academic, I have subsequently been involved in a number of startups, and I have applied machine learning to all of these practical application settings.

Previously, I (Ellen) worked in research laboratories in biochemistry and molecular biology. In addition to having lots of experience with data, I've written extensively on technical subjects. Throughout it all, I've remained fascinated by data and how it speaks to us. I have tried to bring this insight to *Mahout in Action*.

Both of us see that open source only works with input from an active and broad community of participants. A major part of Mahout's success comes from those who have used the software and brought their experience back to the project via discussions in mailing lists, bug fixes, and suggestions.

For this reason, *Mahout in Action* not only provides useful explanations of code, but also guidance regarding the concepts behind the code. This introduction to the framework behind the code will enable you to effectively join in and benefit from the interactive Mahout discussion. We hope this book not only helps the readers of this book, but also helps to expand and enrich Mahout itself.

TED DUNNING AND ELLEN FRIEDMAN

acknowledgments

This book wouldn't be here without the efforts of many people. The authors gratefully acknowledge some of the many here, in no particular order.

- The researchers who have published key papers in the field of machine learning, elaborated on in appendix C
- Mahout users who have spent their time trying beta software, finding and fixing bugs, and providing patches and even suggestions
- Mahout committers, who have dedicated their time to growing, improving, and promoting Mahout
- Manning Publications, which has invested considerable time and effort in bringing this book to market—particularly Katharine Osborne, Karen Tegtmeyer, Jeff Bleiel, Andy Carroll, Melody Dolab, and Dottie Marsico who have been closely involved in creating the final pages you read
- The reviewers who provided valuable feedback during the writing process: Philipp K. Janert, Andrew Oswald, John Griffin, Justin Tyler Wiley, Deepak Vohra, Grant Ingersoll, Isabel Drost, Kenneth DeLong, Eric Raymond, David Grossman, Tom Morton, and Rick Wagner
- Alex Ott who did a thorough technical review of the final manuscript shortly before it went to press
- Manning Early Access (MEAP) readers who posted comments in the Author Online forum
- Everybody who asked questions on the Mahout mailing lists
- Family and friends who supported us through the many hours of writing!

about this book

You may be wondering—is this a book for me?

If you are seeking a textbook on machine learning, no. This book does not attempt to fully explain the theory and derivation of the various algorithms and techniques presented here. Some familiarity with machine learning techniques and related concepts, like matrix and vector math, is useful in reading this book, but not assumed.

If you are developing modern, intelligent applications, then the answer is, yes. This book provides a practical rather than a theoretical treatment of these techniques, along with complete examples and recipes for solutions. It develops some insights gleaned by experienced practitioners in the course of demonstrating how Mahout can be deployed to solve problems.

If you are a researcher in artificial intelligence, machine learning, and related areas—yes. Chances are your biggest obstacle is translating new algorithms into practice. Mahout provides a fertile framework and collection of patterns and ready-made components for testing and deploying new large-scale algorithms. This book is an express ticket to deploying machine learning systems on top of complex distributed computing frameworks.

If you are leading a product team or startup that will leverage machine learning to create a competitive advantage, then yes, this book is also for you. Through real-world examples, it will plant ideas about the many ways these techniques can be deployed. It will also help your scrappy technical team jump directly to a cost-effective implementation that can handle volumes of data previously only realistic for organizations with large technology resources.

Roadmap

This book is divided into three parts, covering collaborative filtering, clustering, and classification in Apache Mahout, respectively.

First, chapter 1 introduces Apache Mahout as a whole. This chapter will get you set up for all of the chapters that follow.

Part 1, which includes chapters 2 through 6, is presented by Sean Owen; it covers collaborative filtering and recommendation. Chapter 2 gives you a first chance to try a Mahout-based recommender engine and evaluate its performance. Chapter 3 discusses how you can represent the data that recommenders use in an efficient way. Then, chapter 4 presents all of the recommender algorithms available in Mahout and compares their strengths and weaknesses. Given that background, chapter 5 presents a case study in which you'll apply the recommender implementations introduced in chapter 4 to a real-world problem, adapt to some particular properties of the data, and create a production-ready recommender engine. Chapter 6 then introduces Apache Hadoop and gives you a first look at machine learning algorithms in a distributed environment by studying a recommender engine based on Hadoop.

Part 2 of the book, including chapters 7 through 12, explores clustering algorithms in Apache Mahout. With the techniques described in this part by Robin Anil, you can group together similar-looking pieces of data into a set or a cluster. Clustering helps uncover interesting groups of information in a large volume of data. This part begins with simple problems in clustering, with examples written in Java. It then introduces more real-world examples and shows how you can make Apache Mahout run as Hadoop jobs that can cluster large amounts of data easily.

Finally, in part 3, Ted Dunning and Ellen Friedman explore classification with Mahout in chapters 13 through 17. You will first learn how to build and train a classifier model by "teaching" an algorithm with a series of examples. Then you will learn how to evaluate and fine tune a classifier's model to give better answers. This part concludes with a real-world case study of classification in action.

Code conventions and downloads

Source code in this book is printed in a `monospaced` font, called out in listings, and annotated with notes about important points. The code listings are intended to be brief and show only essentials. They will not generally show Java imports, class declarations, Java annotations, and other elements that are not essential to the discussion of the code.

Class names in this book are generally printed in a `monospaced` font, inline with the text, to indicate they are classes that can be located and studied within the Apache Mahout source code. For example, `LogLikelihoodSimilarity` is a Java class in Mahout.

Some listings show commands that can be executed. These are written for Unix-like environments such as Mac OS X and Linux distributions. They should work on Microsoft Windows if executed through the Unix-like Cygwin environment.

Compilable copies of the source code in key listings throughout the book are available for download from the publisher's website at www.manning.com/MahoutinAction. These are standalone Java source files and do not include a build script. For simplicity, they can be unpacked and added into a copy of the complete Mahout source distribution under the examples/src/java/main directory. The existing Mahout build environment will then be able to compile the code automatically.

Multimedia extras

All four authors have recorded audio and video segments that accompany specific sections in most of the chapters and provide additional information on selected topics. These segments can be activated in the ebook version of *Mahout in Action*, which is available for free for all owners of the print book, or you can access them for free from the publisher's website at www.manning.com/MahoutinAction/extras. On the printed pages, audio and video icons indicate the topics covered and who is speaking in each segment. Please refer to a full list of these extras that begins on page xxiii.

Author Online

The purchase of *Mahout in Action* includes free access to a private forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and other users. You can access and subscribe to the forum at www.manning.com/MahoutinAction. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct in the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It isn't a commitment to any specific amount of participation on the part of the authors, whose contributions to the book's forum remain voluntary (and unpaid). We suggest you try asking the authors some challenging questions, lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about multimedia extras

Accompanying specific sections in this book are multimedia extras, which are available from www.manning.com/MahoutinAction/extras/ and are free for anyone to listen to or view. Audio or video icons in the margins, like the ones below, indicate which sections of the book have these additional features.


Audio icon


Video icon

about the cover illustration

On the cover of *Mahout in Action* is "A man from Rakov-Potok," a village in northern Croatia. The illustration is taken from a reproduction of an album of Croatian traditional costumes from the mid-nineteenth century by Nikola Arsenovic, published by the Ethnographic Museum in Split, Croatia, in 2003. The illustrations were obtained from a helpful librarian at the Ethnographic Museum in Split, itself situated in the Roman core of the medieval center of the town: the ruins of Emperor Diocletian's retirement palace from around AD 304. The book includes finely colored illustrations of figures from different regions of Croatia, accompanied by descriptions of the costumes and of everyday life.

Rakov-Potok is a picturesque village in the fertile valley of the Sava River in the foothills of the Samobor Mountains, not far from the city of Zagreb. The area has a rich history and you can come across many castles, churches, and ruins that date back to medieval and even Roman times. The figure on the cover is wearing white woolen trousers and a white woolen jacket, richly embroidered in red and blue—a typical costume for the mountaineers of this region.

Dress codes and lifestyles have changed over the last 200 years, and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone of different hamlets or towns separated by only a few miles. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by illustrations from old books and collections like this one.

1 Meet Apache Mahout

This chapter covers

- What Apache Mahout is, and where it came from
- A glimpse of recommender engines, clustering, and classification in the real world
- Setting up Mahout

As you may have guessed from the title, this book is about putting a particular tool, Apache Mahout, to effective use in real life. It has three defining qualities.

First, Mahout is an open source *machine learning* library from Apache. The algorithms it implements fall under the broad umbrella of *machine learning* or *collective intelligence*. This can mean many things, but at the moment for Mahout it means primarily recommender engines (collaborative filtering), clustering, and classification.

It's also *scalable*. Mahout aims to be the machine learning tool of choice when the collection of data to be processed is very large, perhaps far too large for a single machine. In its current incarnation, these scalable machine learning implementations in Mahout are written in Java, and some portions are built upon Apache's Hadoop distributed computation project.

Finally, it's a *Java library*. It doesn't provide a user interface, a prepackaged server, or an installer. It's a framework of tools intended to be used and adapted by developers.

To set the stage, this chapter will take a brief look at the sorts of machine learning that Mahout can help you perform on your data—using recommender engines, clustering, and classification—by looking at some familiar real-world instances.

In preparation for hands-on interaction with Mahout throughout the book, you'll also step through some necessary setup and installation.

1.1 Mahout's story

First, some background on Mahout itself is in order. You may be wondering how to pronounce *Mahout*: in the way it's commonly Anglicized, it should rhyme with *trout*. It's a Hindi word that refers to an elephant driver, and to explain that one, here's a little history.

Mahout began life in 2008 as a subproject of Apache's Lucene project, which provides the well-known open source search engine of the same name. Lucene provides advanced implementations of search, text mining, and information-retrieval techniques. In the universe of computer science, these concepts are adjacent to machine learning techniques like clustering and, to an extent, classification. As a result, some of the work of the Lucene committers that fell more into these machine learning areas was spun off into its own subproject. Soon after, Mahout absorbed the Taste open source collaborative filtering project.

No. 1

Figure 1.1 shows some of Mahout's lineage within the Apache Software Foundation. As of April 2010, Mahout became a top-level Apache project in its own right, and got a brand-new elephant rider logo to boot.

Much of Mahout's work has been not only implementing these algorithms conventionally, in an efficient and scalable way, but also converting some of these algorithms to work at scale on top of Hadoop. Hadoop's mascot is an elephant, which at last explains the project name!

Mahout incubates a number of techniques and algorithms, many still in development or in an experimental phase (https://cwiki.apache.org/confluence/display/MAHOUT/Algorithms). At this early stage in the project's life, three core themes are evident: recommender engines (collaborative filtering), clustering, and classification. This is by no means all that exists within Mahout, but they are the most prominent and mature themes at the time of writing. These, therefore, are the focus of this book.

Chances are that if you're reading this, you're already aware of the interesting potential of these three families of techniques. But just in case, read on.



Figure 1.1 Apache Mahout and its related projects within the Apache Software Foundation

1.2 Mahout's machine learning themes

Although Mahout is, in theory, a project open to implementations of all kinds of machine learning techniques, it's in practice a project that focuses on three key areas of machine learning at the moment. These are recommender engines (collaborative filtering), clustering, and classification.

1.2.1 Recommender engines

Recommender engines are the most immediately recognizable machine learning technique in use today. You'll have seen services or sites that attempt to recommend books or movies or articles based on your past actions. They try to infer tastes and preferences and identify unknown items that are of interest:

- Amazon.com is perhaps the most famous e-commerce site to deploy recommendations. Based on purchases and site activity, Amazon recommends books and other items likely to be of interest. See figure 1.2.
- Netflix similarly recommends DVDs that may be of interest, and famously offered a $1,000,000 prize to researchers who could improve the quality of their recommendations.
- Dating sites like Líbímseti (discussed later) can even recommend people to people.
- Social networking sites like Facebook use variants on recommender techniques to identify people most likely to be as-yet-unconnected friends.

As Amazon and others have demonstrated, recommenders can have concrete commercial value by enabling smart cross-selling opportunities. One firm reports that recommending products to users can drive an 8 to 12 percent increase in sales.[1]

1.2.2 Clustering

Clustering is less apparent, but it turns up in equally well-known contexts. As its name implies, clustering techniques attempt to group a large number of things together into clusters that share some similarity. It's a way to discover hierarchy and order in a

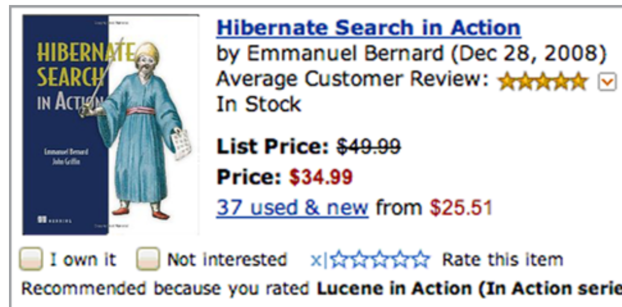


Figure 1.2 A recommendation from Amazon. Based on past purchase history and other activity of customers like the user, Amazon considers this to be something the user is interested in. It can even list similar items that the user has bought or liked that in part caused the recommendation.

[1] Practical eCommerce, "10 Questions on Product Recommendations," http://mng.bz/b6A5



Figure 1.3 A sample news grouping from Google News. A detailed snippet from one representative story is displayed, and links to a few other similar stories within the cluster for this topic are shown. Links to all the stories that are clustered together in this topic are available too.

large or hard-to-understand data set, and in that way reveal interesting patterns or make the data set easier to comprehend.

- Google News groups news articles by topic using clustering techniques, in order to present news grouped by logical story, rather than presenting a raw listing of all articles. Figure 1.3 illustrates this.
- Search engines like Clusty group their search results for similar reasons.
- Consumers may be grouped into segments (clusters) using clustering techniques based on attributes like income, location, and buying habits.

Clustering helps identify structure, and even hierarchy, among a large collection of things that may be otherwise difficult to make sense of. Enterprises might use this technique to discover hidden groupings among users, or to organize a large collection of documents sensibly, or to discover common usage patterns for a site based on logs.

1.2.3 Classification

Classification techniques decide how much a thing is or isn't part of some type or category, or how much it does or doesn't have some attribute. Classification, like clustering, is ubiquitous, but it's even more behind the scenes. Often these systems learn by reviewing many instances of items in the categories in order to deduce classification rules. This general idea has many applications:

- Yahoo! Mail decides whether or not incoming messages are spam based on prior emails and spam reports from users, as well as on characteristics of the email itself. A few messages classified as spam are shown in figure 1.4.
- Google's Picasa and other photo-management applications can decide when a region of an image contains a human face.
- Optical character recognition software classifies small regions of scanned text into individual characters.
- Apple's Genius feature in iTunes reportedly uses classification to classify songs into potential playlists for users.

Classification helps decide whether a new input or thing matches a previously observed pattern or not, and it's often used to classify behavior or patterns as unusual. It could be used to detect suspicious network activity or fraud. It might be used to figure out when a user's message indicates frustration or satisfaction.

Spam (49) Empty		Hevnerco	DishView	Wed 10/28, 12:34 PM
Trash Empty		Customer Service	FINAL NOTIFICATION:..Please r...	Wed 10/28, 4:53 AM
Contacts Add		MmddDdhbh	From: MmddDdhb Read The File.	Wed 10/28, 12:58 AM

Figure 1.4 Spam messages as detected by Yahoo! Mail. Based on reports of email spam from users, plus other analysis, the system has learned certain attributes that usually identify spam. For example, messages mentioning "Viagra" are frequently spam—as are those with clever misspellings like "v1agra." The presence of such terms is an example of an attribute that a spam classifier can learn.

Each of these techniques works best when provided with a large amount of good input data. In some cases, these techniques must not only work on large amounts of input, but must produce results quickly, and these factors make scalability a major issue. And, as mentioned before, one of Mahout's key reasons for being is to produce implementations of these techniques that do scale up to huge input.

1.3 Tackling large scale with Mahout and Hadoop

How real is the problem of scale in machine learning algorithms? Let's consider the size of a few problems where you might deploy Mahout.

Consider that Picasa may have hosted over half a billion photos even three years ago, according to some crude estimates.[2] This implies millions of new photos per day that must be analyzed. The analysis of one photo by itself isn't a large problem, even though it's repeated millions of times. But the learning phase can require information from each of the billions of photos simultaneously—a computation on a scale that isn't feasible for a single machine.

According to a similar analysis, Google News sees about 3.5 million new news articles *per day*. Although this does not seem like a large amount in absolute terms, consider that these articles must be clustered, along with other recent articles, in *minutes* in order to become available in a timely manner.

The subset of rating data that Netflix published for the Netflix Prize contained 100 million ratings. Because this was just the data released for contest purposes, presumably the total amount of data that Netflix actually has and must process to create recommendations is many times larger!

Machine learning techniques must be deployed in contexts like these, where the amount of input is large—so large that it isn't feasible to process it all on one computer, even a powerful one. Without an implementation such as Mahout, these would be impossible tasks. This is why Mahout makes scalability a top priority, and why this book will focus, in a way that others don't, on dealing with large data sets effectively.

Sophisticated machine learning techniques, applied at scale, were until recently only something that large, advanced technology companies could consider using. But today computing power is cheaper than ever and more accessible via open source frameworks like Apache's Hadoop. Mahout attempts to complete the puzzle by

[2] *Google Blogoscoped*, "Overall Number of Picasa Photos" (March 12, 2007), http://blogoscoped.com/archive/2007-03-12-n67.html

providing quality, open source implementations capable of solving problems at this scale with Hadoop, and putting this into the hands of all technology organizations.

Some of Mahout makes use of Hadoop, which includes an open source, Java-based implementation of the MapReduce distributed computing framework popularized and used internally at Google (http://labs.google.com/papers/mapreduce.html). MapReduce is a programming paradigm that at first sounds odd, or too simple to be powerful. The MapReduce paradigm applies to problems where the input is a set of key-value pairs. A *map* function turns these key-value pairs into other intermediate key-value pairs. A *reduce* function merges in some way all values for each intermediate key to produce output. Actually, many problems can be framed as MapReduce problems, or as a series of them. The paradigm also lends itself quite well to parallelization: all of the processing is independent and so can be split across many machines. Rather than reproduce a full explanation of MapReduce here, we refer you to tutorials such as the one provided by Hadoop (http://hadoop.apache.org/mapreduce/docs/current/mapred_tutorial.html).

Hadoop implements the MapReduce paradigm, which is no small feat, even given how simple MapReduce sounds. It manages storage of the input, intermediate key-value pairs, and output; this data could potentially be massive and must be available to many worker machines, not just stored locally on one. It also manages partitioning and data transfer between worker machines, as well as detection of and recovery from individual machine failures. Understanding how much work goes on behind the scenes will help prepare you for how relatively complex using Hadoop can seem. It's not just a library you add to your project. It's several components, each with libraries and (several) standalone server processes, which might be run on several machines. Operating processes based on Hadoop isn't simple, but investing in a scalable, distributed implementation can pay dividends later: your data may quickly grow to great size, and this sort of scalable implementation is a way to future-proof your application.

In chapter 6, this book will try to cut through some of that complexity to get you running on Hadoop quickly, after which you can explore the finer points and details of operating full clusters and tuning the framework. Because this complex framework that needs a great deal of computing power is becoming so popular, it's not surprising that cloud computing providers are beginning to offer Hadoop-related services. For example, Amazon offers Elastic MapReduce (http://aws.amazon.com/elasticmapreduce/), a service that manages a Hadoop cluster, provides the computing power, and puts a friendlier interface on the otherwise complex task of operating and monitoring a large-scale job with Hadoop.

1.4 Setting up Mahout

You'll need to assemble some tools before you can play along at home with the code we'll present in the coming chapters. We assume you're comfortable with Java development already.

Mahout and its associated frameworks are Java-based and therefore platform-independent, so you should be able to use it with any platform that can run a modern JVM. At times, we'll need to give examples or instructions that will vary from platform to platform. In particular, command-line commands are somewhat different in a Windows shell than in a FreeBSD tcsh shell. We'll use commands and syntax that work with bash, a shell found on most Unix-like platforms. This is the default on most Linux distributions, Mac OS X, many Unix variants, and Cygwin (a popular Unix-like environment for Windows). Windows users who wish to use the Windows shell are the most likely to be inconvenienced by this. Still, it should be simple to interpret and translate the listings given in this book to work for that shell.

1.4.1 Java and IDEs

Java is likely already installed on your personal computer if you've done any Java development so far. Note that Mahout requires Java 6. If you're not sure which Java version you have, open a terminal and type `java -version`. If the reported version doesn't begin with 1.6, you need to also install Java 6.

Windows and Linux users can find a Java 6 JVM from Oracle at http://www.oracle.com/technetwork/java/. Apple provides a Java 6 JVM for Mac OS X 10.5 and 10.6. In Mac OS X, if it doesn't appear that Java 6 is being used, open the Java Preferences application under the /Applications/Utilities folder. This will allow you to select Java 6 as the default.

Most people will find it quite a bit easier to edit, compile, and run this book's examples with the help of an IDE; this is *strongly* recommended. Eclipse (http://www.eclipse.org) is the most popular, free Java IDE. Installing and configuring Eclipse is beyond the scope of this book, but you should spend some time becoming familiar with it before proceeding. NetBeans (http://netbeans.org/) is also a popular, free IDE. IntelliJ IDEA (http://www.jetbrains.com/idea/index.html) is another powerful and popular IDE, and a free community version is now available.

As an example of what you can do with an IDE, IDEA can create a new project from an existing Maven model; if you specify the root directory of the Mahout source code upon creating a project, it will automatically configure and present the entire project in an organized manner. It's then possible to drop the source code found throughout this book into the examples/src/main/java/ source root and run it from within IDEA with one click—the details of dependencies and compilation are managed automatically. This should prove far easier than attempting to compile and run the code manually.

> **NOTE** If the test program uses a file with input data, it usually should run in the same directory as the file with the data. Consult your IDE's guide for details about how to set up a working directory for each of the examples.

1.4.2 Installing Maven

As with many Apache projects, Mahout's build and release system is built around Maven (http://maven.apache.org). Maven is a command-line tool that manages dependencies, compiles code, packages releases, generates documentation, and publishes formal releases. Although it has some superficial resemblance to the also-popular Ant build tool, it isn't the same. Ant is a flexible, lower-level scripting language, and Maven is a higher-level tool more purpose-built for dependency and release management. Because Mahout uses Maven, you should install Maven yourself.

Mac OS X users will be pleased to find that Maven should already be installed. If not, install Apple's Developer Tools. Type `mvn --version` on the command line. If you successfully see a version number, and the version is at least 2.2, you're ready to go. If not, you should install a local copy of Maven.

Users of Linux distributions with a decent package management system may be able to use it to quickly obtain a recent version of Maven. Otherwise, standard procedure would be to download a binary distribution, unpack it to a common location, such as /usr/local/maven, and then edit bash's configuration file, ~/.bashrc, to include a line like `export PATH=/usr/local/maven/bin:$PATH`. This will ensure that the `mvn` command is always available.

If you're using an IDE like Eclipse or IntelliJ, it already includes Maven integration. Refer to its documentation to learn how to enable the Maven integration. This will make working with Mahout in an IDE much simpler, as the IDE can use the project's Maven configuration file (pom.xml) to instantly configure and import the project.

> **NOTE** For Eclipse, you'll need to install the m2eclipse plugin (http://www.eclipse.org/m2e/). For NetBeans, Maven support is available out of the box starting with version 6.7; for previous versions, you'll need to install a separate plugin.

1.4.3 Installing Mahout

Mahout is still in development, and this book was written to work with the 0.5 release of Mahout. This release and others may be downloaded by following the instructions at https://cwiki.apache.org/confluence/display/MAHOUT/Downloads; the archive of the source code may be unpacked anywhere that's convenient on your computer.

Because Mahout is changing frequently, and bug fixes and improvements are added regularly, it may be useful to use a later release than 0.5 (or even the latest unreleased code from Subversion; see https://cwiki.apache.org/confluence/display/MAHOUT/Version+Control). Future point releases should be backwards compatible with the examples in this book.

Once you've obtained the source, either from Subversion or from a release archive, create a new project for Mahout in your IDE. This is IDE-specific; refer to its documentation for particulars of how this can be accomplished. It will be easiest to use your IDE's Maven integration to import the Maven project from the pom.xml file in the root of the project source.

Once all of these setup steps are completed, you can easily create a new source directory within this project to hold the sample code that will be introduced in upcoming chapters. With the project properly configured, you should be able to compile and run the code transparently with no further effort.

The source code for the examples is available from Manning's site (http://www.manning.com/MahoutinAction/) or from GitHub (https://github.com/tdunning/MiA). Use the instructions provided with the source code to set up your work environment.

1.4.4 Installing Hadoop

For some activities later in this book, you'll need your own local installation of Hadoop. You don't need a cluster of computers to run Hadoop. Setting up Hadoop isn't difficult, but it's not trivial. Rather than repeat the procedures, we'll direct you to obtain a copy of Hadoop version 0.20.2 from the Hadoop website at http://hadoop.apache.org/common/releases.html, and then set up Hadoop for pseudo-distributed operation by following the Single Node Setup documentation (http://hadoop.apache.org/common/docs/current/single_node_setup.html).

1.5 Summary

Mahout is a young, open source, scalable machine learning library from Apache, and this book is a practical guide to using Mahout to solve real problems with machine learning techniques. In particular, you'll soon explore recommender engines, clustering, and classification. If you're a researcher familiar with machine learning theory and you're looking for a practical how-to guide, or you're a developer looking to quickly learn best practices from practitioners, this book is for you.

These techniques are no longer merely theory. We've already noted some well-known examples of recommender engines, clustering, and classification deployed in the real world: e-commerce, email, videos, photos, and more involve large-scale machine learning. These techniques have been deployed to solve real problems and even generate value for enterprises—and they're now accessible via Mahout.

We've also noted the vast amount of data sometimes employed with these techniques—scalability is a uniquely persistent concern in this area. We took a first look at MapReduce and Hadoop and how they power some of the scalability that Mahout provides.

Because this will be a hands-on, practical book, we've set you up to begin working with Mahout right away. At this point, you should have assembled the tools you'll need to work with Mahout and be ready for action. Because this book is intended to be practical, let's wrap up the opening remarks now and get on to some real code with Mahout. Read on!

Part 1

Recommendations

This first part of the book, including chapters 2 through 6, explores one of the three pillars of Apache Mahout's machine learning implementations: collaborative filtering and recommendation. With these techniques, you can understand a person's tastes and find new, desirable content for them automatically. This part is also a warm-up for the rest of the book, which will depend heavily on the Apache Hadoop distributed computing framework. You'll meet machine learning in Apache Mahout in simple Java first, and then in Hadoop.

Chapter 2 introduces recommender engines, as implemented in Mahout, and covers evaluating performance in the context of a runnable example. Chapter 3 discusses representing recommender data efficiently in Mahout. Chapter 4 catalogs the various recommender engine implementations available in Mahout and their varying features and attributes.

Chapter 5 presents a case study based on data from a dating site, which shows how you can adapt the approaches in Mahout to cope with real-world data and produce a production-ready recommender. Finally, chapter 6 provides a first look at Mahout's use of Apache Hadoop to implement a large-scale distributed recommender engine.

2 Introducing recommenders

This chapter covers

- What recommenders are, within Mahout
- A first look at a recommender in action
- Evaluating the accuracy and quality of recommender engines
- Evaluating a recommender on a real data set: GroupLens

Each day we form opinions about things we like, don't like, and don't even care about. It happens unconsciously. You hear a song on the radio and either notice it because it's catchy, or because it sounds awful—or maybe you don't notice it at all. The same thing happens with T-shirts, salads, hairstyles, ski resorts, faces, and television shows.

Although people's tastes vary, they do follow patterns. People tend to like things that are similar to other things they like. Because Sean loves bacon-lettuce-and-tomato sandwiches, you can guess that he would enjoy a club sandwich, which is

mostly the same sandwich but with turkey. Likewise, people tend to like things that similar people like.

These patterns can be used to predict such likes and dislikes. Recommendation is all about predicting these patterns of taste, and using them to discover new and desirable things you didn't already know about.

After introducing the idea of recommendation in more depth, this chapter will help you experiment with some Mahout code to run a simple recommender engine and understand how well it works, in order to give you an immediate feel for how Mahout works in this regard.

2.1 Defining recommendation

You picked up this book from the shelf for a reason. Maybe you saw it next to other books you know and find useful, and you figured the bookstore put it there because people who like those books tend to like this one too. Maybe you saw this book on the bookshelf of a coworker, who you know shares your interest in machine learning, or perhaps they recommended it to you directly.

These are different, but valid, strategies for discovering new things: to discover items you may like, you could look to what people with similar tastes seem to like. On the other hand, you could figure out what items are like the ones you already like, again by looking to others' apparent preferences. In fact, these are the two broadest categories of recommender engine algorithms: *user-based* and *item-based* recommenders, both of which are well represented within Mahout.

Strictly speaking, the preceding scenarios are examples of *collaborative filtering*—producing recommendations based on, and only based on, knowledge of users' relationships to items. These techniques require no knowledge of the properties of the items themselves. This is, in a way, an advantage. This recommender framework doesn't care whether the items are books, theme parks, flowers, or even other people, because nothing about their attributes enters into any of the input.

There are other approaches based on the attributes of items, and these are generally referred to as *content-based* recommendation techniques. For example, if a friend recommended this book to you because it's a Manning book, and the friend likes other Manning books, then the friend is engaging in something more like content-based recommendation. The suggestion is based on an attribute of the books: the publisher.

There's nothing wrong with content-based techniques; on the contrary, they can work quite well. They're necessarily domain-specific approaches, and they'd be hard to meaningfully codify into a framework. To build an effective content-based book recommender, one would have to decide which attributes of a book—page count, author, publisher, color, font—are meaningful, and to what degree. None of this knowledge translates into any other domain; recommending books this way doesn't help in the area of recommending pizza toppings.

For this reason, Mahout won't have much to say about content-based recommendations. These ideas *can* be built into, and on top of, what Mahout provides; as such, it

might technically be called a *collaborative filtering framework*. An example of this will follow in chapter 5, where you'll build a recommender for a dating site.

For now, though, it's time to experiment with collaborative filtering within Mahout by creating some simple input and finding recommendations based on the input.

2.2 Running a first recommender engine

Mahout contains a recommender engine—several types of them, in fact, beginning with conventional user-based and item-based recommenders. It includes implementations of several other algorithms as well, but for now we'll explore a simple user-based recommender.

2.2.1 Creating the input

To explore recommendations in Mahout, it's best to start with a trivial example.

The recommender requires input—data on which it will base its recommendations. This data takes the form of *preferences* in Mahout-speak. Because the recommender engines that are most familiar involve recommending items to users, it'll be most convenient to talk about preferences as associations from users to items—though, as noted previously, these users and items could be anything. A preference consists of a user ID and an item ID, and usually a number expressing the strength of the user's preference for the item. IDs in Mahout are always numbers—integers, in fact. The preference value could be anything, as long as larger values mean stronger positive preferences. For instance, these values might be ratings on a scale of 1 to 5, where *1* indicates items the user can't stand, and *5* indicates favorites.

Create a text file containing data about users, who are cleverly named "1" to "5," and their preferences for seven books, simply called "101" through "107." In real life, these might be customer IDs and product IDs from a company database; Mahout doesn't require that the users and items be named with numbers. Write this data in a simple comma-separated value format.

Copy the following example into a file and save it as intro.csv.

Listing 2.1 Recommender input file, intro.csv

```
1,101,5.0
1,102,3.0
1,103,2.5

2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0

3,101,2.5
3,104,4.0
3,105,4.5
3,107,5.0

4,101,5.0
4,103,3.0
```

User ID, item ID, preference value

User 1 has preference 3.0 for item 102

```
4,104,4.5
4,106,4.0

5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

With some study, trends appear. Users 1 and 5 seem to have similar tastes. They both like book 101, like 102 a little less, and like 103 less still. The same goes for users 1 and 4, as they seem to like 101 and 103 identically (no word on how user 4 likes 102 though). On the other hand, users 1 and 2 have tastes that seem to run counter to each other—1 likes 101 whereas 2 doesn't, and 1 likes 103 but 2 is just the opposite. Users 1 and 3 don't overlap much—the only book both express a preference for is 101. Figure 2.1 illustrates the relationships, both positive and negative, between users and items.

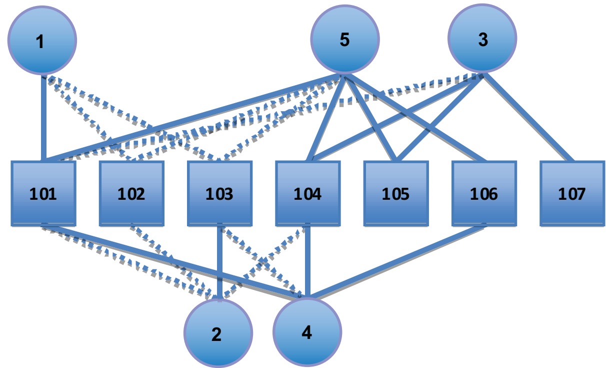


Figure 2.1 Relationships between users 1 to 5 and items 101 to 107. Dashed lines represent associations that seem negative—the user doesn't seem to like the item much but expresses a relationship to the item.

2.2.2 Creating a recommender

So what book might you recommend to user 1? Not 101, 102, or 103—user 1 already knows about these books, apparently, and recommendation is about discovering new things. Intuition suggests that because users 4 and 5 seem similar to 1, it would be good to recommend something that user 4 or user 5 likes. That leaves books 104, 105, and 106 as possible recommendations. On the whole, 104 seems to be the most liked of these possibilities, judging by the preference values of 4.5 and 4.0 for item 104.

Now, run the following code.

Listing 2.2 A simple user-based recommender program with Mahout

```
class RecommenderIntro {

  public static void main(String[] args) throws Exception {

    DataModel model =
      new FileDataModel (new File("intro.csv"));          <- Load data file

    UserSimilarity similarity =
      new PearsonCorrelationSimilarity (model);
    UserNeighborhood neighborhood =
```

```
      new NearestNUserNeighborhood (2, similarity, model);

    Recommender recommender = new GenericUserBasedRecommender (
        model, neighborhood, similarity);              <- Create recommender engine

    List<RecommendedItem> recommendations =
        recommender.recommend(1, 1);                   <- For user 1, recommend 1 item

    for (RecommendedItem recommendation : recommendations) {
      System.out.println(recommendation);
    }

  }

}
```

To help visualize the relationship between these basic components, see figure 2.2. Not all Mahout-based recommenders will look like this—some will employ different components with different relationships. But this gives a sense of what's going on in the example.

We'll discuss each of these components in more detail in the next two chapters, but we can summarize the role of each component now. A `DataModel` implementation stores and provides access to all the preference, user, and item data needed in the computation. A `UserSimilarity` implementation provides some notion of how similar two users are; this could be based on one of many possible metrics or calculations. A `UserNeighborhood` implementation defines a notion of a group of users that are most similar to a given user. Finally, a `Recommender` implementation pulls all these components together to recommend items to users.

2.2.3 *Analyzing the output*

When you run the code in listing 2.2, the output in your terminal or IDE should be as follows:

```
RecommendedItem [item:104, value:4.257081]
```

The request asked for one top recommendation, and it got one. The recommender engine recommended book 104 to user 1. Further, it said that the recommender engine did so because it estimated user 1's preference for book 104 to be about 4.3, and that was the highest among all the items eligible for recommendation.

That's not bad. Book 107 did not appear; it was also recommendable but was only associated with a user who had different tastes. It picked 104 over 106, and this makes

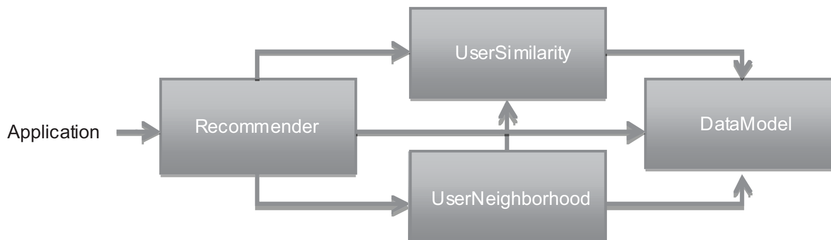


Figure 2.2 Simplified illustration of component interaction in a Mahout user-based recommender

sense when you note that 104 is a bit more highly rated overall. Further, the output contained a reasonable estimate of how much user 1 likes item 104—something between the 4.0 and 4.5 that users 4 and 5 expressed.

The right answer isn't obvious from looking at the data, but the recommender engine made some decent sense of it and returned a defensible answer. If you got a pleasant tingle out of seeing this simple program give a useful and non-obvious result from a small pile of data, the world of machine learning is for you!

For clear, small data sets, producing recommendations is as trivial as it appears in the preceding example. In real life, data sets are huge, and they're noisy. For example, imagine a popular news site recommending news articles to readers. Preferences are inferred from article clicks, but many of these preferences may be bogus—maybe a reader clicked on an article but didn't like it, or clicked on the wrong story. Perhaps many of the clicks occurred while not logged in, so they can't be associated with a user. And imagine the size of the data set—perhaps billions of clicks in a month.

Producing the *right* recommendations from this data and producing them *quickly* isn't trivial. Later, in case studies, we'll present the tools Mahout provides to attack a range of such problems. They'll show how standard approaches can produce poor recommendations or take a great deal of memory and CPU time, and how to configure and customize Mahout to improve performance.

2.3 Evaluating a recommender

A recommender engine is a tool, a means to answer the question, "What are the best recommendations for a user?" Before investigating the *answers*, it's best to investigate the *question*. What exactly is a good recommendation? And how does one know when a recommender is producing them? The remainder of this chapter pauses to explore how to evaluate a recommender, because this will be useful when looking at specific recommender systems.

The best possible recommender would be a sort of psychic that could somehow know, before you do, exactly how much you'd like every possible item that you've not yet seen or expressed any preference for. A recommender that could predict all your preferences exactly would present all items ranked by your future preference and be done. These would be the best possible recommendations.

And indeed, most recommender engines operate by trying to do just this, estimating ratings for some or all other items. As a result, one way of evaluating a recommender's recommendations is to evaluate the quality of its estimated preference values—that is, by evaluating how closely the estimated preferences match the actual preferences.

2.3.1 Training data and scoring

Those actual preferences don't exist, though. Nobody knows for sure how you'll like some new item in the future (including you). In the context of a recommender engine, this can be *simulated* by setting aside a small part of the *real* data set as *test* data. These test preferences aren't present in the training data fed into a recommender

engine under evaluation. Instead, the recommender is asked to estimate preferences for the missing test data, and estimates are compared to the actual values.

From there, it's fairly simple to produce a kind of score for the recommender. For example, it's possible to compute the average difference between estimate and actual preference. With a score of this type, *lower* is better, because that would mean the estimates differed from the actual preference values by less. A score of 0.0 would mean perfect estimation—no difference at all between estimates and actual values.

Sometimes the root-mean-square of the differences is used: this is the square root of the average of the *squares* of the differences between actual and estimated preference values. See table 2.1 for an illustration of this. Again, lower is better.

Table 2.1 An illustration of the average difference and root-mean-square calculation

	Item 1	Item 2	Item 3
Actual	3.0	5.0	4.0
Estimate	3.5	2.0	5.0
Difference	0.5	3.0	1.0
Average difference	= (0.5 + 3.0 + 1.0) / 3 = 1.5		
Root-mean-square	$= \sqrt{((0.5^2 + 3.0^2 + 1.0^2) / 3)} = 1.8484$		

Table 2.1 shows the difference between a set of actual and estimated preferences, and how they're translated into scores. Root-mean-square more heavily penalizes estimates that are way off, as with item 2 here, and that's considered desirable by some. For example, an estimate that's off by two whole stars is probably more than twice as bad as one off by just one star. Because the simple average of differences is perhaps more intuitive and easier to understand, upcoming examples will use it.

2.3.2 *Running RecommenderEvaluator*

Let's revisit the example code and evaluate the simple recommender on this simple data set, as shown in the following listing.

Listing 2.3 Configuring and running an evaluation of a recommender

```
RandomUtils.useTestSeed();
DataModel model = new FileDataModel (new File("intro.csv"));

RecommenderEvaluator evaluator =
  new AverageAbsoluteDifferenceRecommenderEvaluator ();

RecommenderBuilder builder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(DataModel model)
      throws TasteException {
```

Generates repeatable results (annotation for the first two lines)

Builds recommender in listing 2.2 (annotation for the RecommenderBuilder lines)

```
    UserSimilarity similarity = new PearsonCorrelationSimilarity (model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood (2, similarity, model);
    return
      new GenericUserBasedRecommender (model, neighborhood, similarity);
  }
};
double score = evaluator.evaluate(
    builder, null, model, 0.7, 1.0);
System.out.println(score);
```

Trains with 70% of data; tests with 30%

Most of the action happens in `evaluate()`: the `RecommenderEvaluator` splits the data into a training and test set, builds a new training `DataModel` and `Recommender` to test, and compares its estimated preferences to the actual test data.

Note that there's no `Recommender` passed to `evaluate()`. That's because, inside, the method will need to build a `Recommender` around a newly created training `DataModel`. The caller must provide an object that can build a `Recommender` from a `DataModel`—a `RecommenderBuilder`. Here, the method builds the same implementation that you tried earlier in this chapter.

2.3.3 Assessing the result

The program in listing 2.3 prints the result of the evaluation: a score indicating how well the `Recommender` performed. In this case, you should simply see `1.0`. Even though a lot of randomness is used inside the evaluator to choose test data, the result should be consistent because of the call to `RandomUtils.useTestSeed()`, which forces the same random choices each time. This is only used in such examples, and unit tests, to guarantee repeatable results. *Don't use it in your real code!*

What this value means depends on the implementation used—here, `AverageAbsoluteDifferenceRecommenderEvaluator`. A result of 1.0 from this implementation means that, on average, the recommender estimates a preference that deviates from the actual preference by 1.0.

A value of 1.0 isn't great, on a scale of 1 to 5, but there's very little data here to begin with. Your results may differ because the data set is split randomly, and the training and test set may differ with each run.

This technique can be applied to any `Recommender` and `DataModel`. To use root-mean-square scoring, replace `AverageAbsoluteDifferenceRecommenderEvaluator` with the implementation `RMSRecommenderEvaluator`.

Also, instead of passing the `null` parameter to `evaluate()`, you could instead pass an instance of `DataModelBuilder`, which can be used to control how the training `DataModel` is created from the training data. Normally the default is fine; it may not be if you're using a specialized implementation of `DataModel` in your deployment—a `DataModelBuilder` is how you would inject it into the evaluation process.

The `1.0` parameter passed to `evaluate()` at the end controls how much of the overall input data is used. Here, it means 100 percent. This parameter can be used to produce a quicker, if less accurate, evaluation by using only a little of a potentially

huge data set. For example, `0.1` would mean 10 percent of the data is used and 90 percent is ignored. This is quite useful when you want to quickly test small changes to a `Recommender`.

2.4 Evaluating precision and recall

We could also take a broader view of the recommender problem: it's not strictly *necessary* to estimate preference values in order to produce recommendations. In many cases, presenting an ordered list of recommendations, from best to worst, is sufficient, without including estimated preference values. In fact, in some cases the exact ordering of the list doesn't matter much—a set of a few good recommendations is fine.

Taking this more general view, we could also apply classic information retrieval metrics to evaluate recommenders: precision and recall. These terms are typically applied to things like search engines, which return a set of best results for a query out of many possible results.

A search engine shouldn't return irrelevant results in the top results, but it should strive to return as many relevant results as possible. *Precision* is the proportion of top results that are relevant, for some definition of relevant. *Precision at 10* would be this proportion judged from the top 10 results. *Recall* is the proportion of all relevant results included in the top results. See figure 2.3 for a visualization of these ideas.

These terms can easily be adapted to recommenders: *precision* is the proportion of top recommendations that are good recommendations, and *recall* is the proportion of good recommendations that appear in top recommendations. The next section will define *good*.

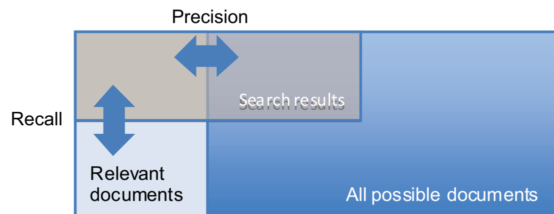


Figure 2.3 An illustration of precision and recall in the context of search results

2.4.1 Running RecommenderIRStatsEvaluator

Again, Mahout provides a fairly simple way to compute these values for a `Recommender`, as showing in the next listing.

Listing 2.4 Configuring and running a precision and recall evaluation

```
RandomUtils.useTestSeed();
DataModel model = new FileDataModel (new File("intro.csv"));

RecommenderIRStatsEvaluator evaluator =
  new GenericRecommenderIRStatsEvaluator ();
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(DataModel model)
      throws TasteException {
    UserSimilarity similarity = new PearsonCorrelationSimilarity (model);
    UserNeighborhood neighborhood =
```

```
      new NearestNUserNeighborhood (2, similarity, model);
    return
      new GenericUserBasedRecommender (model, neighborhood, similarity);
  }
};
IRStatistics stats = evaluator.evaluate(
    recommenderBuilder, null, model, null, 2,
    GenericRecommenderIRStatsEvaluator.CHOOSE_THRESHOLD,
    1.0);
```

Evaluate precision and recall at 2

```
System.out.println(stats.getPrecision());
System.out.println(stats.getRecall());
```

Without the call to `RandomUtils.useTestSeed()`, the result you see would vary significantly due to the random selection of training data and test data, and because the data set is so small here. But with the call, the result ought to be

```
0.75
1.0
```

Precision at 2 is 0.75; on average about three-quarters of recommendations were good. Recall at 2 is 1.0; all good recommendations are among those recommended.

But what exactly is a good recommendation here? The framework was asked to decide—it didn't receive a definition. Intuitively, the most highly preferred items in the test set are the good recommendations, and the rest aren't.

Listing 2.5 User 5's preferences in the test data set

```
5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

Look at user 5 in this simple data set again. Let's imagine the preferences for items 101, 102, and 103 were withheld as test data. The preference values for these are 4.0, 3.0, and 2.0. With these values missing from the training data, a recommender engine ought to recommend 101 before 102, and 102 before 103, because this is the order in which user 5 prefers these items. But would it be a good idea to recommend 103? It's last on the list; user 5 doesn't seem to like it much. Book 102 is just average. Book 101 looks reasonable, because its preference value is well above average. Maybe 101 is a good recommendation; 102 and 103 are *valid*, but not *good* recommendations.

And this is the thinking that the `RecommenderEvaluator` employs. When not given an explicit threshold that divides good recommendations from bad, the framework will pick a threshold, per user, that's equal to the user's average preference value, μ, plus one standard deviation, σ:

$$\text{threshold} = \mu + \sigma$$

If you've forgotten your statistics, don't worry. This takes items whose preference value isn't merely a little more than average (μ), but above average by a significant amount (σ). In practice, this means that about 16 percent of items that are most highly preferred are considered good recommendations to make back to the user. The other arguments to this method are similar to those discussed before and are more fully documented in the project's Javadoc.

2.4.2 Problems with precision and recall


No. 2

The usefulness of precision and recall tests in the context of recommenders depends entirely on how well a good recommendation can be defined. In the preceding section, the threshold was either given or defined by the framework. A poor choice will hurt the usefulness of the resulting score.

There's a subtler problem with these tests, though. Here, they necessarily pick the set of good recommendations from among items for which the user has already expressed some preference. But the best recommendations are not necessarily among those the user already knows about!

Imagine running such a test for a user who would absolutely love the little-known French cult film, *My Brother the Armoire*. Let's say it's objectively a great recommendation for this user, but the user has never heard of this movie. If a recommender actually returned this film when recommending movies, it would be penalized; the test framework can only pick good recommendations from among those in the user's set of preferences already.

The issue is further complicated when the preferences are Boolean and contain no preference value. There isn't even a notion of relative preference on which to select a subset of good items. The best the test can do is randomly select some preferred items as the good ones.

The test nevertheless has some use. The items a user prefers are a reasonable proxy for the best recommendations for the user, but they're by no means a perfect one. In the case of Boolean preference data, only a precision-recall test is available anyway. It's worth understanding the test's limitations in this context.

2.5 Evaluating the GroupLens data set

With these tools in hand, we can discuss not only the speed, but also the quality, of recommender engines. Although examples with *large* amounts of real data are still a couple of chapters away, it's already possible to quickly evaluate performance on a small data set.

2.5.1 Extracting the recommender input

GroupLens (http://grouplens.org/) is a research project that provides several data sets of different sizes, each derived from real users' ratings of movies. It's one of several large, real-world data sets available, and more will be explored in this book.

From the GroupLens site, locate and download the "100K data set," currently accessible at http://www.grouplens.org/node/73. Unarchive the file you download, and within find the file called ua.base. This is a tab-delimited file with user IDs, item IDs, ratings (preference values), and some additional information.

Will this file work? Tabs, not commas, separate its fields, and it includes an extra field of information at the end as well. Yes, the file will work with `FileDataModel` as is. Return to the code in listing 2.3, which built a `RecommenderEvaluator`, and try passing in the location of ua.base instead of the small data file. Run it again. This time, the evaluation should take a couple of minutes, because it's now based on 100,000 preference values instead of a handful.

At the end, you should get a number around 0.9. That's not bad, but somehow being off by almost a whole point on a scale of 1 to 5 doesn't sound great. Perhaps the particular `Recommender` implementation we're using isn't the best for this kind of data?

2.5.2 Experimenting with other recommenders

Let's test-drive a *slope-one* recommender on this data set—a simple algorithm that will reappear in chapter 4. It's as easy as replacing the `RecommenderBuilder` with one that uses `org.apache.mahout.cf.taste.impl.recommender.slopeone.SlopeOneRecommender`, as shown in the next listing.

Listing 2.6 Changing the evaluation program to run a `SlopeOneRecommender`

```
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(DataModel model)
      throws TasteException {
    return new SlopeOneRecommender(model);
  }
};
```

Run the evaluation again. You should find that it's much quicker, and it produces an evaluation result around 0.748. That's a move in the right direction.

This isn't to say slope-one is always better or faster. Each algorithm has its own characteristics and properties that can interact in hard-to-predict ways with a given data set. For example, slope-one happens to be quick to compute recommendations at runtime, but it takes significant time to precompute its internal data structures before it can start. The user-based recommender that we tried first could be faster and more accurate on other data sets. We'll explore the relative virtues of each algorithm in chapter 4.

This difference highlights how important testing and evaluation on real data are—and how relatively painless it can be with Mahout.

2.6 Summary

In this chapter, we introduced the idea of recommender engines. We even created some small input for a simple Mahout `Recommender`, ran it through a simple computation, and explained the results.

We then took time to evaluate the quality of a recommender engine's output, because we'll need to do this frequently in the coming chapters. This chapter covered evaluating the accuracy of a `Recommender`'s estimated preferences, as well as traditional precision and recall metrics as applied to recommendations. Finally, we tried evaluating a real data set from GroupLens and observed how evaluations can be used to empirically discover improvements to a recommender engine.

Before we go on to study recommender engines in detail, it's important to spend some time with another foundational concept for recommenders in Mahout: the representation of data. We'll look at that in the next chapter.

3 Representing recommender data

This chapter covers

- How Mahout represents recommender data
- `DataModel` implementations and usage
- Handling data without preference values

The quality of recommendations is largely determined by the quantity and quality of data. "Garbage in, garbage out," has never been more true than here. Having high-quality data is a good thing, and generally, having lots of it is also good.

Recommender algorithms are data-intensive by nature; their computations access a great deal of information. Runtime performance is therefore greatly affected by the quantity of data and its representation. Intelligently choosing data structures can affect performance by orders of magnitude, and, at scale, it matters a lot.

This chapter explores Mahout's key classes for representing and accessing recommender-related data. You'll get a better sense of why users and items, and their associated preferences, are represented the way they are in Mahout for effi-

ciency and scalability. This chapter also looks in detail at the key abstraction in Mahout that provides access to this data: a `DataModel`.

Finally, we look at problems and opportunities that arise when user and item data has no concept of ratings or preferences values—the so-called *Boolean preferences*, which require special handling.

The first section introduces the basic unit of recommender data, a user-item preference.

3.1 Representing preference data

The input to a recommender engine is preference data—who likes what, and how much. That means the input to Mahout recommenders is simply a set of user ID, item ID, and preference value tuples—a large set, of course. Sometimes, even preference values are omitted.

3.1.1 The Preference object

A `Preference` is the most basic abstraction, representing a single user ID, item ID, and preference value. One object represents one user's preference for one item. `Preference` is an interface, and the implementation you're most likely to use is `GenericPreference`. For example, the following line creates a representation of user 123's preference value of 3.0 for item 456:

```
new GenericPreference(123, 456, 3.0f)
```

How is a set of `Preferences` represented? If you gave reasonable answers like `Collection<Preference>` or `Preference[]`, you'd be wrong in most cases in the Mahout APIs. Collections and arrays turn out to be quite inefficient for representing large numbers of `Preference` objects. If you've never investigated the overhead of an `Object` in Java, prepare to be shocked!

A single `GenericPreference` contains 20 bytes of useful data: an 8-byte user ID (Java `long`), an 8-byte item ID (`long`), and a 4-byte preference value (`float`). The object's existence entails a startling amount of overhead: 28 bytes! This includes an 8-byte reference to the object, and, due to `Object` overhead and other alignment issues, another 20 bytes of space within the representation of the object itself. Hence a `GenericPreference` object already consumes 140 percent more memory than it needs to, just due to overhead.

> **NOTE** The actual amount of overhead varies depending on the JVM's implementation; the preceding figures were for Apple's 64-bit Java 6 VM for Mac OS X 10.6.

What can be done? In the recommender algorithms, it's common to need a collection of all preferences associated with one user, or one item. In such a collection, the user ID, or item ID, will be identical for all `Preference` objects, which seems redundant.

3.1.2 PreferenceArray and implementations

Enter `PreferenceArray`, an interface whose implementations represent a collection of preferences with an array-like API. For example, `GenericUserPreferenceArray` represents all preferences associated with one user. Internally, it maintains a single user ID, an array of item IDs, and an array of preference values. The marginal memory required per preference in this representation is only 12 bytes (one 8-byte item ID and a 4-byte preference value in an array). Compare this to the approximately 48 bytes required for a full `Preference` object. The four-fold memory savings alone justifies this special implementation, but it also provides a small performance win, because far fewer objects must be allocated and examined by the garbage collector. Compare figures 3.1 and 3.2 to understand how the savings are accomplished.

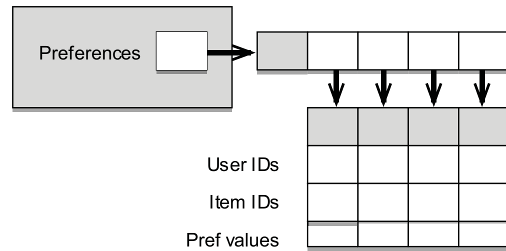


Figure 3.1 A less-efficient representation of preferences using an array of `Preference` objects. Gray areas represent, roughly, the `Object` overhead. White areas are data, including `Object` references.

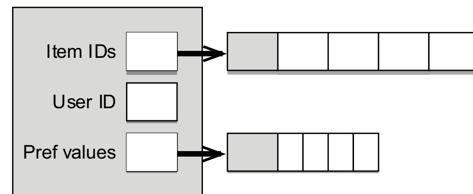


Figure 3.2 A more efficient representation using `GenericUserPreferenceArray`

The following code shows how a `PreferenceArray` is typically constructed and accessed.

Listing 3.1 Setting preference values in a `PreferenceArray`

```
PreferenceArray user1Prefs = new GenericUserPreferenceArray(2);
user1Prefs.setUserID(0, 1L);                       <- Sets user ID for preferences
user1Prefs.setItemID(0, 101L);
user1Prefs.setValue(0, 2.0f);
user1Prefs.setItemID(1, 102L);                     | Expresses preferences
user1Prefs.setValue(1, 3.0f);
Preference pref = user1Prefs.get(1);               <- Materializes Preference for item 102
```

There exists, likewise, an implementation called `GenericItemPreferenceArray`, which encapsulates all preferences associated with an item, rather than with a user. Its purpose and usage are entirely analogous.

3.1.3 Speeding up collections

"Wonderful! Mahout has already reinvented an array of Java objects," you're probably thinking. Well, buckle up, because that's not the end of it. Did we mention that scale is

important? Hopefully you're already persuaded that the amount of data you may face with these techniques is unusually huge, and it may merit unusual responses.

The reduced memory requirement that `PreferenceArray` and its implementations bring is well worth its complexity. Cutting memory requirements by 75 percent isn't just saving a couple of megabytes—it's saving *tens of gigabytes* of memory at reasonable scale. That may be the difference between the data fitting and not fitting on your existing hardware. It may mean the difference between having to invest in a lot more RAM and maybe a new 64-bit system, and not having to. That's a small but real savings.

3.1.4 FastByIDMap and FastIDSet

You won't be surprised to hear that the Mahout recommenders make heavy use of typical data structures like maps and sets, but they don't use the normal Java collections implementations like `TreeSet` and `HashMap`. Instead, throughout the implementation and API you'll find `FastMap`, `FastByIDMap`, and `FastIDSet`. These are something like `Map` and `Set`, but they're specialized explicitly and only for what Mahout recommenders need. They reduce memory footprint rather than significantly increasing performance.

None of this should be construed as a criticism of the Java Collections framework. On the contrary, collections are well designed for their purpose of being effective in a wide range of contexts. They can't make many assumptions about usage patterns. Mahout's needs are much more specific, and stronger assumptions about usage are available. These are the key differences:

- Like `HashMap`, `FastByIDMap` is hash-based. It uses linear probing, rather than separate chaining, to handle hash collisions. This avoids the need for an additional `Map.Entry` object per entry; as discussed, `Objects` consume a surprising amount of memory.
- Keys and members are always long primitives in Mahout recommenders, not `Objects`. Using long keys saves memory and improves performance.
- The `Set` implementation isn't implemented using a `Map` underneath.
- `FastByIDMap` can act like a cache, because it has a notion of maximum size; beyond this size, infrequently used entries will be removed when new ones are added.

The storage difference is significant: `FastIDSet` requires about 14 bytes per member on average, compared to 84 bytes for `HashSet`. `FastByIDMap` consumes about 28 bytes per entry, compared to again about 84 bytes per entry for `HashMap`. It goes to show that when one can make stronger assumptions about usage, significant improvements are possible—here, largely in memory requirements. Given the volume of data involved in recommender systems, these custom implementations more than justify themselves.

So, where are these clever classes used?

3.2 In-memory DataModels

The abstraction that encapsulates recommender input data in Mahout is `DataModel`. Implementations of `DataModel` provide efficient access to data required by various recommender algorithms. For example, a `DataModel` can provide a count or list of all user IDs in the input data, or provide all preferences associated with an item, or provide a count of all users who express a preference for a set of item IDs.

This section focuses on some of the highlights; a more detailed account of `DataModel`'s API can be found in the online Javadoc documentation (https://builds.apache.org/job/Mahout-Quality/javadoc/).

3.2.1 GenericDataModel

The simplest `DataModel` implementation available is an in-memory implementation, `GenericDataModel`. It's appropriate when you want to construct your data representation in memory, programmatically, rather than base it on an existing external source of data, such as a file or relational database. It simply accepts preferences as inputs, in the form of a `FastByIDMap` that maps user IDs to `PreferenceArrays` with data for those users, as shown in the next listing.

Listing 3.2 Defining input data programmatically with `GenericDataModel`

```
FastByIDMap<PreferenceArray> preferences =
  new FastByIDMap<PreferenceArray>();
PreferenceArray prefsForUser1 = new GenericUserPreferenceArray(10);

prefsForUser1.setUserID(0, 1L);
prefsForUser1.setItemID(0, 101L);
prefsForUser1.setValue(0, 3.0f);
prefsForUser1.setItemID(1, 102L);
prefsForUser1.setValue(1, 4.5f);
... (8 more)

preferences.put(1L, prefsForUser1);

DataModel model = new GenericDataModel(preferences);
```

Add first of 10 preferences

Attach user 1's preference to input

How much memory does a `GenericDataModel` use? The number of preferences stored dominates memory consumption. Some empirical testing reveals that it consumes about 28 bytes of Java heap space per preference. This includes all data *and* other supporting data structures, like indexes. You can try this if you like: load a `GenericDataModel`, call `System.gc()` a few times, and then compare the results of `Runtime.totalMemory()` and `Runtime.freeMemory()`. This is crude, but it should give a reasonable estimate of how much memory the data is consuming.

3.2.2 File-based data

You won't typically use `GenericDataModel` directly. Instead, you'll likely encounter it via `FileDataModel`, which reads data from a file and stores the resulting preference data in memory, in a `GenericDataModel`.

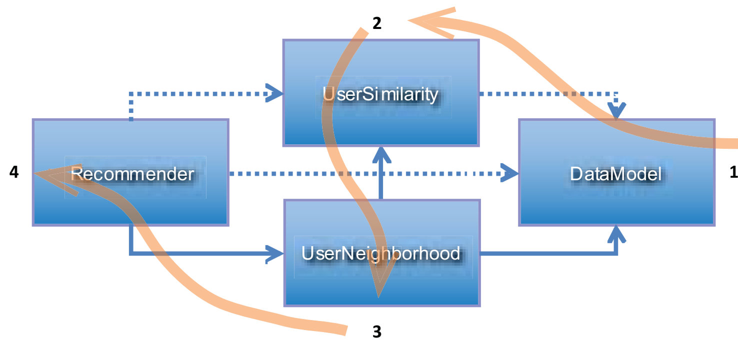


Figure 3.3 An illustration of dependencies in a simple user-based recommender system, and the order in which components refresh their data structures

Just about any reasonable file will do, such as the simple comma-separated value file from chapter 2. Each line contained one datum: user ID, item ID, and preference value. Tab-separated files will work too. Zipped and gzipped files will also work, if their names end in *.zip* or *.gz*, respectively. It's a good idea to store this data in a compressed format, because it can be huge, and it compresses well.

3.2.3 Refreshable components

While we're talking about loading data, it's useful to talk about *reloading* data, and the `Refreshable` interface, which several components in the Mahout recommender-related classes implement. It exposes a single method, `refresh(Collection<Refreshable>)`. This method simply requests that the component reload, recompute, and refresh its own state based on the latest input data available, after asking its dependencies to do likewise.

For example, a `Recommender` will likely call `refresh()` on the `DataModel` on which it's based before recomputing its own internal indexes of the data. Cyclical dependencies and shared dependencies are managed intelligently, as illustrated in figure 3.3 (based on figure 2.2, from the last chapter).

Note that `FileDataModel` will only reload data from the underlying file when asked to do so. It won't automatically detect updates or regularly attempt to reload the file's contents, for performance reasons. This is what the `refresh()` method is for. You probably don't want to cause only the `FileDataModel` to refresh, but also any objects that depend on its data. That's why, in practice, you'll almost surely call `refresh()` on a `Recommender` as follows.

Listing 3.3 Triggering a refresh of a recommender system

```
DataModel dataModel = new FileDataModel(new File("input.csv");
Recommender recommender = new SlopeOneRecommender(dataModel);
...
recommender.refresh(null);                 <- Refresh DataModel, then itself
```

Because scale is a pervasive theme of this book, we should emphasize another useful feature of `FileDataModel`: update files. Data changes, and usually the data that changes is only a tiny subset of all the data—maybe even just a few new data points, in comparison to a billion existing ones. Pushing around a brand new copy of a file containing a billion preferences, just to push a few updates, is wildly inefficient.

3.2.4 Update files

`FileDataModel` supports update files. These are just more data files that are read after the main data file, and that overwrite any previously read data. New preferences are added and existing ones are updated. Deletes are handled by providing an empty preference value string.

For example, consider the following update file:

```
1,108,3.0
1,103,
```

This says, "update (or create) user 1's preference for item 108, and set the value to 3.0" and "remove user 1's preference for item 103."

These update files must simply exist in the same directory as the main data file, and their names must begin with the same prefix, up to the first period. For example, if the main data file is foo.txt.gz, the update files might be named foo.1.txt.gz and foo.2.txt.gz. Yes, they may be compressed.

3.2.5 Database-based data

Sometimes data is just too large to fit into memory. Once the data set is several tens of millions of preferences, memory requirements grow to several gigabytes, and this amount of memory may be unavailable in some contexts.

It's possible to store and access preference data from a relational database; Mahout supports this. Several classes in Mahout's recommender implementation will attempt to push computations into the database for performance.

Note that running a recommender engine from data in a database will be much slower, by orders of magnitude, than using in-memory data representations. It's no fault of the database; properly tuned and configured, a modern database is excellent at indexing and retrieving information efficiently, but the overhead of retrieving, marshalling, serializing, transmitting, and deserializing result sets is still much greater than the overhead of reading data from optimized in-memory data structures. This adds up quickly for recommender algorithms, which are data intensive. Yet, working from a database may be desirable in cases where you have no choice, or where the data set isn't huge and reusing an existing table of data is desirable for integration purposes.

3.2.6 JDBC and MySQL

Preference data is accessed via JDBC, using implementations of `JDBCDataModel`. At the moment, the primary subclass of `JDBCDataModel` is one written for use with

MySQL 5.x: `MySQLJDBCDataModel`. It may well work with older versions of MySQL, or even other databases, because it tries to use standard ANSI SQL where possible. It isn't difficult to create variations, as needed, to use database-specific syntax and features.

> **NOTE** There's an implementation of `JDBCDataModel` for PostgreSQL in the development version of Mahout. There's also a `GenericJDBCDataModel` class that allows you to use data from databases that have no specialized implementation.

By default, the implementation assumes that all preference data exists in a table called taste_preferences, with a column for user IDs named user_id, a column for item IDs named item_id, and column for preference values named preference. This schema is illustrated in table 3.1. This table could also contain a field called timestamp, whose type should be compatible with Java's `long` type.

Table 3.1 Illustration of default table schema for taste_preferences in MySQL

user_id	item_id	preference
BIGINT NOT NULL INDEX	BIGINT NOT NULL INDEX	FLOAT NOT NULL
PRIMARY KEY		

3.2.7 *Configuring via JNDI*

`JDBCDataModel` `JDBCDataModel` implementations also assume that the database containing this table is accessible via a `DataSource` object registered to JNDI under the name jdbc/taste.

What is JNDI, you may be asking? It's the Java Naming and Directory Interface, and it's a key part of the J2EE (Java 2 Enterprise Edition) specification. If you're using a recommender engine in a web application, and you're using a servlet container like Tomcat or Resin, then you're likely already using JNDI indirectly. If you're configuring your database details through the container (such as through Tomcat's server.xml file), you'll find that this configuration is typically made available as a `DataSource` in JNDI.

You can configure a database as jdbc/taste with details about the database that the `JDBCDataModel` ought to use. Here's a snippet suitable for use with Tomcat.

Listing 3.4 Configuring a JNDI `DataSource` in Tomcat

```
<Resource
  name="jdbc/taste"
  auth="Container"
  type="javax.sql.DataSource"
  username="user"
  password="password"
  driverClassName="com.mysql.jdbc.Driver"
  url="jdbc:mysql://localhost:3306/mydatabase"/>
```

These default names can be overridden to reflect your environment. You don't have to name your database and column exactly as above.

3.2.8 Configuring programmatically

You also don't have to use JNDI directly; you can instead pass a DataSource directly to the MySQLJDBCDataModel constructor. The next listing shows a full example of configuring MySQLJDBCDataModel, including the use of the MySQL Connector/J driver (http://www.mysql.com/products/connector/) and a DataSource with customized table and column names.

Listing 3.5 Configuring a DataSource programmatically

```
MysqlDataSource dataSource = new MysqlDataSource ();
dataSource.setServerName("my_database_host");
dataSource.setUser("my_user");
dataSource.setPassword("my_password");
dataSource.setDatabaseName("my_database_name");
JDBCDataModel dataModel = new MySQLJDBCDataModel(
  dataSource, "my_prefs_table", "my_user_column",
  "my_item_column", "my_pref_value_column");
```

This is all that's needed to use data in a database for recommendations.

You've now got a DataModel compatible with all the recommender components! But as the documentation for MySQLJDBCDataModel makes clear, producing the recommendations efficiently requires properly configuring the database and the driver. In particular,

- The user ID and item ID columns should be non-nullable and must be indexed.
- The primary key must be a composite of user ID and item ID.
- Select data types for the columns that correspond to Java's long and float types. In MySQL, these are BIGINT and FLOAT.
- Look to tuning the buffers and query caches; see the Javadoc for MySQLJDBCDataModel.
- When using MySQL's Connector/J driver, set driver parameters such as cachePreparedStatements to true. Again, see the Javadoc for suggested values.

This covers the basics of working with DataModels in Mahout's recommender engine framework. One significant variant on these implementations should be discussed: representing data when there are no preference values. This may sound strange, because it seems like preference values are the core of the input data required by a recommender engine. But sometimes a *value* isn't present, or ignoring the value is useful.

3.3 Coping without preference values

Sometimes, the preferences that go into a recommender engine have no values. That is, a user and item are associated, but there's no notion of the strength of that association. For example, imagine a news site recommending articles to users based on

previously viewed articles. A view establishes some association between the user and the item, but that's about all that's available. It isn't common for users to rate articles. It's not even common for users to do anything more than view an article. All that's known in this case is which articles the user is associated with, and little more.

In such cases, you have no choice; there's no preference value in the input to begin with. The techniques and advice in the remainder of the chapter still apply to these scenarios. But it's also sometimes advantageous to ignore preference values when they do exist in the input. At least, sometimes, it doesn't hurt.

It's not a question of forgetting all associations between users and items. Rather, the idea is to ignore the purported *strength* of the preference. For example, rather than considering which movies you've seen and how you've rated them when recommending a new movie, you might do as well to simply consider what movies you have seen. Rather than knowing that "user 1 expresses preference 4.5 for movie 103," it could be useful to forget the 4.5 and take, as input, data like "user 1 is associated with movie 103." Figure 3.4 illustrates the difference.

In Mahout-speak, these associations without preference values are called *Boolean preferences*, for lack of a better term, because an association can have one of two values: it exists, or it doesn't exist. This doesn't mean that the data consists of *yes* and *no* preferences for items—that would give three states for every possible user-item association: like, dislike, or nothing at all.

3.3.1 When to ignore values



Why would one ignore preference values? It might be beneficial in a context where liking and not liking an item are relatively similar states, at least when compared with having no association at all.

Let's consider an example. Imagine a fellow who knows he doesn't like the work of the classical composer Rachmaninoff. In fact, in this fellow's iTunes library, he's rated several Rachmaninoff pieces with 1 or 2 stars. There's a vast world of music out there, some of which he's never even heard of (like Norwegian death metal). That he even

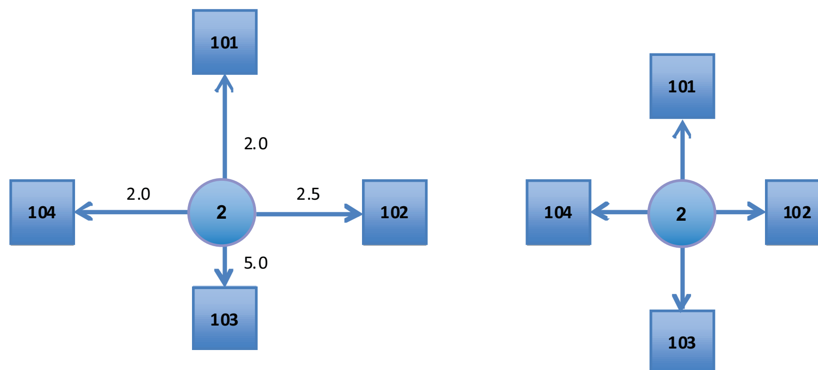


Figure 3.4 An illustration of user relationships to items with preference values (left) and without preference values (right)

knows Rachmaninoff enough to dislike his work, and to even have the pieces in his iTunes library to begin with, indicates an association with this composer, and even a possible preference for things like it. This is significant when considered in comparison to the vast world of things he doesn't even know about. Although he might rate Rachmaninoff a 1 and Brahms a 5 if pressed to do so, in reality these both communicate something similar: an interest in classical music. Forgetting the actual ratings, therefore, reflects that fact and may even make for better recommendations.

You may object that this is the user's fault. Shouldn't he think of Rachmaninoff as a 4, because it's stuff like Norwegian death metal that's conceptually a 1 to him? Maybe so, but that's life. This only underscores the fact that input is often problematic. You may also object that, although this reasoning stands up when recommending music taken from all genres, forgetting this data may be worse when just recommending from classical composers. This is true; a good solution for one domain doesn't always translate to others.

3.3.2 In-memory representations without preference values

Not having preference values dramatically simplifies the representation of preference data, and this enables better performance and significantly lower memory usage. As you've seen, Mahout `Preference` objects store the preference value as a 4-byte float in Java. Not having preference values in memory ought to save 4 bytes per preference value. Indeed, repeating the same rough testing as before shows the overall memory consumption per preference drops by about 4 bytes to 24 bytes on average.

This value comes from testing the twin of `GenericDataModel`, called `GenericBooleanPrefDataModel`. This is another in-memory `DataModel` implementation, but one that internally doesn't store preference values. It simply stores associations as `FastIDSets`—for example, one for each user, to represent the item IDs that that user is associated with. No preference values are included.

Because it's also a `DataModel`, `GenericBooleanPrefDataModel` is a drop-in replacement for `GenericDataModel`. Some methods of `DataModel` will be faster with this new implementation, such as `getItemIDsForUser()`, because the implementation already has this readily available. Some will be slower, such as `getPreferencesFromUser()`, because the new implementation doesn't use `PreferenceArrays` and must materialize one to implement the method.

You may wonder what `getPreferenceValue()` returns, because there are no preference values. It doesn't throw `UnsupportedOperationException`; it returns the same artificial value in all cases: 1.0. This is important to note, because components that rely on a preference value will still get one from this `DataModel`. These preference values are artificial and fixed, which can cause some subtle issues.

Let's observe, by returning to the GroupLens example from the last chapter. Here is the code from listing 3.5, but set up to use a `GenericBooleanPrefDataModel`.

Listing 3.6 Creating and evaluating with Boolean data

```
DataModel model = new GenericBooleanPrefDataModel(
  GenericBooleanPrefDataModel.toDataMap(
    new FileDataModel(new File("ua.base"))));          <- Use GenericBooleanPrefDataModel

RecommenderEvaluator evaluator =
  new AverageAbsoluteDifferenceRecommenderEvaluator();

RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  public Recommender buildRecommender(DataModel model)
      throws TasteException {
    UserSimilarity similarity =
      new PearsonCorrelationSimilarity(model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood(10, similarity, model);
    return
      new GenericUserBasedRecommender(model, neighborhood, similarity);
  }
};

DataModelBuilder modelBuilder = new DataModelBuilder() {
  public DataModel buildDataModel(
      FastByIDMap<PreferenceArray> trainingData) {
    return new GenericBooleanPrefDataModel(
      GenericBooleanPrefDataModel.toDataMap(
        trainingData));                                 <- Build a GenericBooleanPrefDataModel
  }
};

double score = evaluator.evaluate(
  recommenderBuilder, modelBuilder, model, 0.9, 1.0);
System.out.println(score);
```

The twist in this example is the `DataModelBuilder`. You can use it to control how the evaluation process will construct its `DataModel` for training data, rather than let it construct a simple `GenericDataModel`. The `GenericBooleanPrefDataModel` takes its input in a slightly different way—as a bunch of `FastIDSets` rather than `PreferenceArrays`—and the convenience method `toDataMap()` exists to translate between the two.

Before proceeding to the next section, try running this code—it won't complete successfully.

3.3.3 *Selecting compatible implementations*

You should find that running the code in listing 3.6 results in an `IllegalArgumentException` from the `PearsonCorrelationSimilarity` constructor. This may be surprising at first: isn't `GenericBooleanPrefDataModel` also a `DataModel`, and nearly the same as `GenericDataModel` except that it doesn't store distinct preference values?

This similarity metric, like `EuclideanDistanceSimilarity`, refuses to work without preference values because its results would be undefined or meaningless, producing useless results. The Pearson correlation between two data sets will be undefined if the

two data sets are simply the same value, repeated. Here, the `DataModel` pretends that all preference values are 1.0. Similarly, computing a Euclidean distance between users who all correspond to the same point in space—(1.0, 1.0, ..., 1.0)—would be meaningless because all similarities would be 1.0.

> **NOTE** The Pearson correlation is a ratio of the covariance of the two data sets to their standard deviations. When all the data are 1, both of these values are 0, giving a correlation of 0/0, which is certainly "not a number" as far as Java is concerned.

More generally, not every implementation will work well with every other, even though components are implementing a set of standard interfaces for interchangeability. To solve the immediate problem, substitute a suitable similarity metric. `LogLikelihoodSimilarity` is one such implementation, because it isn't based on actual preference values. (We'll discuss these similarity metrics later.) Plug it in, in place of `PearsonCorrelationSimilarity`. The result is 0.0. That's excellent, because it means perfect prediction. Is it too good to be true?

Unfortunately, yes. That's the average difference between estimated and actual preferences in a world where every preference value is 1. *Of course* the result is 0; the test itself is invalid because it will only ever result in 0.

But a precision and recall evaluation is still valid. Let's try it in the next listing.

Listing 3.7 Evaluating precision and recall with Boolean data

```
DataModel model = new GenericBooleanPrefDataModel(
    new FileDataModel(new File("ua.base")));

RecommenderIRStatsEvaluator evaluator =
  new GenericRecommenderIRStatsEvaluator();
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(DataModel model) {
    UserSimilarity similarity = new LogLikelihoodSimilarity(model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood(10, similarity, model);
    return new GenericUserBasedRecommender(
        model, neighborhood, similarity);
  }
};
DataModelBuilder modelBuilder = new DataModelBuilder() {
  @Override
  public DataModel buildDataModel(
      FastByIDMap<PreferenceArray> trainingData) {
    return new GenericBooleanPrefDataModel(
      GenericBooleanPrefDataModel.toDataMap(trainingData));
  }
};
IRStatistics stats = evaluator.evaluate(
    recommenderBuilder, modelBuilder, model, null, 10,
    GenericRecommenderIRStatsEvaluator.CHOOSE_THRESHOLD,
    1.0);
```

```
System.out.println(stats.getPrecision());
System.out.println(stats.getRecall());
```

The result is about 24.7 percent for both precision and recall. That's not great; recall that this means only about 1 in 4 recommendations returned are good, and about 1 in 4 good recommendations are returned.

This is traceable to a third problem; preference values are still lurking in one place here: `GenericUserBasedRecommender`. This recommender still orders its recommendations based on estimate preference, but these values are all 1.0. The ordering is therefore essentially random. Instead, you can introduce `GenericBooleanPrefUserBasedRecommender` (yes, that's about as long as the class names will get). This variant will produce a more meaningful ordering in its recommendations. It weights items that are associated with many other similar users, and with users that are more similar, more heavily. It doesn't produce a weighted average.

Try substituting this implementation and run the code again. The result is 22.9 percent or so—about the same. This strongly suggests that we aren't using a terribly effective recommender system for this data. Our purpose here isn't to fix this, but merely to look at how to effectively deploy Boolean data in Mahout recommenders.

Boolean variants of the other `DataModel`s exist as well. `FileDataModel` will automatically use a `GenericBooleanPrefDataModel` internally if its input data contains no preference values (lines of the form `userID,itemID` only). Similarly, `MySQLBooleanPrefDataModel` is suitable for use with a database table that has no preference value column. It's otherwise entirely analogous to `MySQLJDBCDataModel`. This implementation, in particular, can take advantage of many more shortcuts in the database to improve performance.

Finally, if you're wondering whether you can mix Boolean and non-Boolean data within one `DataModel`, the answer is, no. One solution is to ignore what preference values you have and treat it all as Boolean data. Or, if you don't wish to throw out those preference values for some reason, those missing a preference value can be inferred by some means, even if it's as simple as filling in the average of all existing preference values as a placeholder.

3.4 Summary

In this chapter, we looked at how preference data is represented in a Mahout recommender. This includes `Preference` objects, but also specialized array and collection-like implementations like `PreferenceArray` and `FastByIDMap`. These specializations exist largely to reduce memory usage.

We looked at `DataModel`s, which are the abstraction for recommender input as a whole. `GenericDataModel` stores data in memory, as does `FileDataModel` after reading input from a file. `JDBCDataModel` and other implementations exist to support data based on a relational database table; we examined integration with MySQL in particular.

Finally, we looked at how all this changes when the input data doesn't contain preference values—only user-item associations. Sometimes this is all that's available, and it certainly requires less storage. We looked at incompatibilities between this sort of data and standard components like `PearsonCorrelationSimilarity`. We also looked at how to fix such problems and get a functioning recommender based on Boolean input data.

In the next chapter, we'll move on from examining the possibilities for data representation and look into the available recommender implementations themselves.

4 Making recommendations

This chapter covers

- User-based recommenders, in depth
- Similarity metrics
- Item-based and other recommenders

Having spent the last chapter discussing how to evaluate recommenders and represent the data input to a recommender, it's now time to examine the recommenders themselves in detail. That's where the real action begins.

Previous chapters alluded to two well-known styles of recommender algorithms, both of which are implemented in Mahout: user-based recommenders and item-based recommenders. In fact, you already encountered a user-based recommender in chapter 2. This chapter explores the theory behind these algorithms, as well as the Mahout implementations of both, in detail.

Both algorithms rely on a similarity metric, or notion of sameness between two things, whether they're users or items. There are many ways to define similarity, and this chapter will introduce in detail your choices within Mahout. These include

implementations based on the Pearson correlation, log likelihood, Spearman correlation, Tanimoto coefficient, and more.

Finally, you'll also become familiar with other styles of recommender algorithms implemented within Mahout, including slope-one recommenders, SVD-based recommenders, and clustering-based recommenders.

4.1 Understanding user-based recommendation

If you've seen a recommender algorithm explained before, chances are it was a user-based recommender algorithm. This is the approach described in some of the earliest research in the field, and it's certainly implemented within Mahout. The label *user-based* is somewhat imprecise, as any recommender algorithm is based on user- and item-related data. The defining characteristic of a user-based recommender algorithm is that it's based upon some notion of similarities between users. In fact, you've probably encountered this type of algorithm in everyday life.

4.1.1 When recommendation goes wrong

Have you ever received a CD as a gift? I (Sean) did, as a youngster, from well-meaning adults. One of these adults evidently headed down to the local music store and cornered an employee, where the following scene unfolded:

ADULT: *I'm looking for a CD for a teenager.*

EMPLOYEE: *OK, what does this teenager like?*

ADULT: *Oh, you know, what all the young kids like these days.*

EMPLOYEE: *What kind of music or bands?*

ADULT: *It's all noise to me. I don't know.*

EMPLOYEE: *Uh, well... I guess lots of young people are buying this boy band album here by New 2 Town?*

ADULT: *Sold*!

You can guess the result. Needless to say, they instead give me gift certificates now. Unfortunately, this example of user-based recommendation gone wrong has played out many times. The intuition was sound: because teenagers have relatively related tastes in music, one teenager would be more likely to enjoy an album that other teenagers have enjoyed. Basing recommendations on similarities among people is quite reasonable.

Of course, recommending an album from a band that teenage *girls* swoon over probably isn't the best thing for a teenage *boy*. The error here was that the similarity metric wasn't effective. Yes, teenagers as a group have relatively homogenous tastes: you're more likely to find pop songs than zydeco or classical music. But, the similarity is too weak to be useful: it's not true that teenage *girls* have enough in common with teenage *boys* when it comes to music to form the basis of a recommendation.

4.1.2 When recommendation goes right

Let's rewind the scenario and imagine how it could have gone better:

ADULT: *I'm looking for a CD for a teenage boy.*

EMPLOYEE: *What kind of music or bands does he like?*

ADULT: *I don't know, but his best friend is always wearing a Bowling In Hades T-shirt.*

EMPLOYEE: *Ah yes, a very popular nu-metal band from Cleveland. Well, we do have the new Bowling In Hades best-of album over here,* Impossible Split: The Singles 1997–2000...

That's better. The recommendation was based on the assumption that two good friends share somewhat similar tastes in music. With a reliable similarity metric in place, the outcome is probably better. It's far more likely that these best friends share a love for Bowling In Hades than any two random teenagers. Here's another way it could have gone better:

ADULT: *I'm looking for a CD for a teenage boy.*

EMPLOYEE: *What kind of music or bands does he like?*

ADULT: *"Music?" Ha, well, I wrote down the bands from posters on his bedroom wall. The Skulks, Rock Mobster, the Wild Scallions... mean anything to you?*

EMPLOYEE: *I see. Well, my kid is into some of those albums too. And he won't stop talking about some new album from Diabolical Florist, so maybe...*

Now they've inferred a similarity based directly on tastes in music. Because the two kids in question both prefer some of the same bands, it stands to reason they'll each like a lot in the rest of each other's collections. That's even better reasoning than guessing that their tastes are similar because they're friends. They've actually based their idea of similarity between the two teenagers on observed tastes in music. This is the essential logic of a user-based recommender system.

4.2 Exploring the user-based recommender

If those two adults kept going, they'd further refine their reasoning. Why base the choice of gift on just one other kid's music collection? How about finding several other similar kids? They would pay attention to which kids seemed most similar—those with most of the same posters and T-shirts and CDs scattered on top of stereos—and look at which bands seemed most important to those most similar kids, and figure those would make the best gift. (And then, they might become Mahout users!)

4.2.1 The algorithm

The user-based recommender algorithm comes out of this intuition. It's a process of recommending items to some user, denoted by *u*, as seen below:

```
for every item i that u has no preference for yet
  for every other user v that has a preference for i
    compute a similarity s between u and v
    incorporate v's preference for i, weighted by s, into a running average
return the top items, ranked by weighted average
```

The outer loop simply considers every known item (that the user hasn't already expressed a preference for) as a candidate for recommendation. The inner loop looks

to any other user who has expressed a preference for this candidate item, and sees what their preference value for it was. In the end, the values are averaged to come up with an estimate—a weighted average, that is. Each preference value is weighted in the average by how similar that user is to the target user. The more similar a user, the more heavily their preference value is weighted.

It would be terribly slow to examine every item. In reality, a neighborhood of most similar users is computed first, and only items known to those users are considered:

```
for every other user w
  compute a similarity s between u and w
  retain the top users, ranked by similarity, as a neighborhood n
for every item i that some user in n has a preference for,
      but that u has no preference for yet
  for every other user v in n that has a preference for i
    compute a similarity s between u and v
    incorporate v's preference for i, weighted by s, into a running average
```

The primary difference is that similar users are found first, before seeing what those most-similar users are interested in. Those items become the candidates for recommendation. The rest is the same. This is the standard user-based recommender algorithm, and the way it's implemented in Mahout.

4.2.2 Implementing the algorithm with GenericUserBasedRecommender

The very first recommender example in this book showed a user-based recommender in action in Mahout (listing 2.2). Let's return to it in order to explore the components in use and see how well it performs in the next listing.

Listing 4.1 Revisiting a simple user-based recommender system

```
DataModel model = new FileDataModel(new File("intro.csv"));
UserSimilarity similarity = new PearsonCorrelationSimilarity (model);
UserNeighborhood neighborhood =
  new NearestNUserNeighborhood (2, similarity, model);
Recommender recommender =
  new GenericUserBasedRecommender(model, neighborhood, similarity);
```

`UserSimilarity` encapsulates some notion of similarity among users, and `UserNeighborhood` encapsulates some notion of a group of most-similar users. These are necessary components of the standard user-based recommender algorithm.

There isn't only one possible notion of similarity—the previous CD-choosing dialogue illustrated a few real-world ideas of similarity. There are also many ways to define a neighborhood of most similar users: the 5 most similar? 20? Users with a similarity above a certain value? To illustrate these alternatives, imagine you're creating a guest list for your wedding. You want to invite your closest friends and family to this special occasion, but you have far more friends and family than your budget will allow. Would you decide who is and isn't invited by picking a size first—say 50 people—and picking your 50 closest friends and family? Is 50 the right number, or 40 or 100? Or

would you invite everyone who you consider close? Should you only invite your really close friends? Which one will give the best wedding party? This is analogous to the decision you make when deciding how to pick a neighborhood of similar users.

Plug in new ideas of similarity, and the results change considerably. You can begin to see that there isn't merely one way to produce recommendations—and this is still looking at adjusting one facet of one approach. Mahout isn't a single recommender engine, but an assortment of components that can be plugged together and customized to create an ideal recommender for a particular domain. Here the following components are assembled:

- Data model, implemented via `DataModel`
- User-user similarity metric, implemented via `UserSimilarity`
- User neighborhood definition, implemented via `UserNeighborhood`
- Recommender engine, implemented via a `Recommender` (here, `GenericUserBasedRecommender`)

Getting good results, and getting them fast, is inevitably a long process of experimentation and refinement.

4.2.3 Exploring with GroupLens

Let's return to the GroupLens data set and up the ante with 100 times more data. Return to http://grouplens.org and download the 10 million rating MovieLens data set, which is currently available at http://www.grouplens.org/node/73. Unpack it locally and locate the ratings.dat file inside.

For whatever reason, the format of this data is different from the 100,000 rating data set you looked at before. Whereas its ua.base file was ready for use with `FileDataModel`, this data set's ratings.dat file isn't. It would be simple to use standard command-line text-processing utilities to convert it to comma-separated form, and in general, this is the best approach. Writing custom code to convert the file format, or using a custom `DataModel`, is tedious and error prone.

Luckily, in this particular case there's an easier solution: Mahout's `examples` module includes the custom implementation `GroupLensDataModel`, which extends `FileDataModel` to read this file. Make sure you've included the code under the examples/src/java/main directory in your project in your IDE. Then, swap out `FileDataModel` for the alternative shown in the following listing.

Listing 4.2 Updating listing 4.1 to use a custom `DataModel` for GroupLens

```
DataModel model = new GroupLensDataModel(new File("ratings.dat"));
UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
UserNeighborhood neighborhood =
  new NearestNUserNeighborhood(100, similarity, model);
Recommender recommender =
  new GenericUserBasedRecommender(model, neighborhood, similarity);
LoadEvaluator.runLoad(recommender);
```

Run this code, and the first thing you'll likely encounter is an `OutOfMemoryError`. Ah, a first sighting of issues of scale. By default, Java won't grow its heap past a certain modest size. The heap space available to Java must increase.

This is a good opportunity to discuss what can be done to improve performance by tuning the JVM. Refer to appendix A at this point for a more in-depth discussion of JVM tuning.

4.2.4 Exploring user neighborhoods

Let's next evaluate the recommender's accuracy. Listing 4.3 shows the boilerplate evaluation code one more time; going forward, we'll assume you've got the hang of it and can construct and run evaluations on your own.

Now, let's look at possibilities for configuring and modifying the neighborhood implementation, as shown in the next listing. Remember, this is using 100 times more data as well.

Listing 4.3 Running an evaluation on the simple recommender

```
DataModel model = new GroupLensDataModel (new File("ratings.dat"));
RecommenderEvaluator evaluator =
  new AverageAbsoluteDifferenceRecommenderEvaluator ();
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(
        DataModel model) throws TasteException {
    UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood(100, similarity, model);
    return new GenericUserBasedRecommender(
        model, neighborhood, similarity);
  }
};
double score = evaluator.evaluate(
    recommenderBuilder, null, model, 0.95, 0.05);
System.out.println(score);
```

Note how the final parameter to `evaluate()` is 0.05. This means only 5 percent of all the data is used for evaluation. This is purely for convenience; evaluation is a time-consuming process, and using this full data set could take hours to complete. For purposes of quickly evaluating changes, it's convenient to reduce this value. But using too little data might compromise the accuracy of the evaluation results. The parameter 0.95 simply says to build a model to evaluate with 95 percent of the data, and then test with the remaining 5 percent.

After running this, your evaluation result will vary, but it will likely be around 0.89.

4.2.5 Fixed-size neighborhoods

At the moment, the recommendations from the code in listing 4.3 are derived from a neighborhood of the 100 most similar users (`NearestNUserNeighborhood` was set to a

neighborhood size of 100). The decision to use the 100 users whose similarity is greatest in order to make recommendations is arbitrary. What if this was 10? Recommendations would be based on fewer similar users but would exclude some less-similar users from consideration. See the visualization of a nearest-3 neighborhood in figure 4.1.

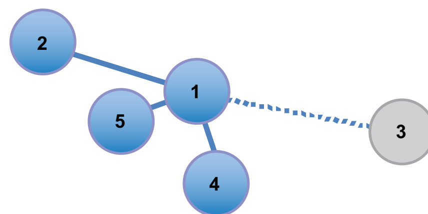


Figure 4.1 An illustration of defining a neighborhood of most similar users by picking a fixed number of closest neighbors. Here, distance illustrates similarity: farther means less similar. A neighborhood around user 1 is chosen to consist of the three most similar users: 5, 4, and 2.

Try replacing 100 with 10. The result of the evaluation, the average difference between estimated and actual preference values, is 0.98 or so. Recall that larger evaluation values are worse, so that's a move in the wrong direction. The most likely explanation is that 10 users is too few. It's likely that the eleventh and twelfth most similar users, and so on, add value. They're still quite similar, and they're associated with items that the first 10 most similar users weren't.

Try a neighborhood of 500 users; the result drops to 0.75, which is of course better. You could evaluate many values and figure out the optimal setting for this data set, but the lesson is that there's no magic value; some experimentation with real data is necessary to tune your recommender.

4.2.6 Threshold-based neighborhood

What if you don't want to build a neighborhood of the *n* most similar users, but rather try to pick the pretty similar users and ignore everyone else? It's possible to pick a similarity threshold and take any users that are at least that similar. Figure 4.2 illustrates a threshold-based definition of user neighborhood; contrast this with the fixed-size neighborhood in figure 4.1.

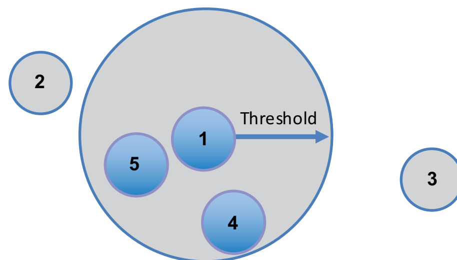


Figure 4.2 Defining a neighborhood of most-similar users with a similarity threshold

The threshold should be between –1 and 1, because all similarity metrics return similarity values in this range. At the moment, our example uses a standard Pearson correlation as the similarity metric. Those familiar with this correlation would likely agree that a value of 0.7 or above is a high correlation and constitutes a sensible definition of pretty similar. Let's switch to use `ThresholdUserNeighborhood`. It's as simple as changing one line to instead instantiate a `ThresholdUserNeighborhood`:

```
new ThresholdUserNeighborhood(0.7, similarity, model)
```

Now the evaluator scores the recommender at 0.84. What if the neighborhood were more selective, using a threshold of 0.9? The score worsens to 0.92; it's likely that the same explanation applies as before—the neighborhood contains too few users at

this threshold. How about 0.5? The score improves to 0.78. Examples that follow will use a threshold-based neighborhood with a threshold of 0.5.

Again, you'd likely want to explore many more values on real data to determine an optimum, but our simple experimentation has already improved estimation accuracy by about 15 percent.

4.3 Exploring similarity metrics

Another important part of user-based recommenders is the `UserSimilarity` implementation. A user-based recommender relies most of all on this component. Without a reliable and effective notion of which users are similar to others, this approach falls apart. The same is true of the user-based recommender's cousin, the item-based recommender, which also relies on similarity. This component is so important that we spend almost a third of this chapter investigating standard similarity metrics and how they're implemented in Mahout.

4.3.1 Pearson correlation–based similarity

So far, examples have used the `PearsonCorrelationSimilarity` implementation, which is a similarity metric based on the Pearson correlation.

The Pearson correlation is a number between –1 and 1 that measures the tendency of two series of numbers, paired up one-to-one, to move together. That is to say, it measures how likely a number in one series is to be relatively large when the corresponding number in the other series is high, and vice versa. It measures the tendency of the numbers to move together *proportionally*, such that there's a roughly linear relationship between the values in one series and the other. When this tendency is high, the correlation is close to 1. When there appears to be little relationship at all, the value is near 0. When there appears to be an opposing relationship—one series' numbers are high exactly when the other series' numbers are low—the value is near –1.

> **NOTE** For statistically inclined readers, the Pearson correlation of two series is the ratio of their covariance to the product of their variances. Covariance is a measure of how much two series move together in absolute terms; it's big when the series moves far in the same direction from their means in the same places. Dividing by the variances merely normalizes for the absolute size of their changes. It's not essential to understand this and other alternative definitions in order to use the Pearson correlation within Mahout, but there's lots of information on the web if you're interested.

This concept, widely used in statistics, can be applied to users to measure their similarity. It measures the tendency of two users' *preference values* to move together—to be relatively high, or relatively low, on the same items. For an example, let's look again at the first sample data file we considered, intro.csv, as follows.

Listing 4.4 A simple recommender input file

```
1,101,5.0
1,102,3.0
1,103,2.5

2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0

3,101,2.5
3,104,4.0
3,105,4.5
3,107,5.0

4,101,5.0
4,103,3.0
4,104,4.5
4,106,4.0

5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

We noted that users 1 and 5 seem similar because their preferences seem to run together. On items 101, 102, and 103, they roughly agree: 101 is the best, 102 somewhat less good, and 103 isn't desirable. By the same reasoning, users 1 and 2 aren't so similar. Note that our reasoning about user 1 can't really include items 104 through 106, because user 1's preferences for 104 through 106 aren't known. The similarity computation can only operate on items that both users have expressed a preference for. (In section 4.3.9, we'll look at what happens when missing preference values like this are *inferred*.)

The Pearson correlation captures these notions, as can be seen in table 4.1. The details of the computation aren't reproduced here; refer to online sources such as http://www.socialresearchmethods.net/kb/statcorr.php for an explanation of how the correlation is computed.

Table 4.1 The Pearson correlation between user 1 and other users based on the three items that user 1 has in common with the others

	Item 101	Item 102	Item 103	Correlation with user 1
User 1	5.0	3.0	2.5	1.000
User 2	2.0	2.5	5.0	-0.764
User 3	2.5	-	-	-
User 4	5.0	-	3.0	1.000
User 5	4.0	3.0	2.0	0.945

Note A user's Pearson correlation with itself is always 1.0.

4.3.2 Pearson correlation problems

Although the results are indeed intuitive, the Pearson correlation has some quirks in the context of recommender engines.

First, it doesn't take into account the number of items in which two users' preferences overlap, which is probably a weakness in the context of recommender engines. Two users that have seen 200 of the same movies, for instance, even if they don't often agree on ratings, are probably more similar than two users who have only ever seen 2 movies in common. This issue appears in a small way in the preceding data; note that users 1 and 5 have both expressed preferences for all three items and seem to have similar tastes. Yet, users 1 and 4 have a higher correlation of 1.0, based on only two overlapping items. This seems a bit counterintuitive.

Second, if two users overlap on only one item, no correlation can be computed because of how the computation is defined. This is why no correlation can be computed between users 1 and 3. This could be an issue for small or sparse data sets, in which users' item sets rarely overlap. Or, one could also view it as a benefit: two users who overlap for only one item are, intuitively, not very similar anyway.

Finally, the correlation is also undefined if either series of preference values are all identical. It doesn't require that *both* series have all the same preference values. For example, if user 5 had expressed a preference of 3.0 for all three items, it wouldn't be possible to compute a similarity between user 1 and user 5 because the Pearson correlation would be undefined, even though user 1 has preferences besides 3.0. This is likewise most probably an issue when users rarely overlap with others in the items they've expressed any preference for.


No. 4

Although the Pearson correlation commonly appears in recommenders in early research papers,[1] and it appears in introductory books on recommenders, it isn't necessarily a good first choice. It's not necessarily bad, either; you simply need to understand how it works.

4.3.3 Employing weighting

`PearsonCorrelationSimilarity` provides an extension to the standard computation, called *weighting*, that mitigates one of the preceding issues.

The Pearson correlation doesn't reflect, directly, the number of items over which it's computed, and for our purposes, that would be useful. When based on more information, the resulting correlation would be a more reliable result. In order to reflect this, it's desirable to push positive correlation values toward 1.0 and negative values toward –1.0 when the correlation is based on more items. Alternatively, you could imagine pushing the correlation values towards some mean preference value when the correlation is based on fewer items; the effect would be similar, but the

[1] Appendix C lists some of these papers. See in particular Breese, Heckerman, and Kadie, "Empirical Analysis of Predictive Algorithms for Collaborative Filtering" and Herlocker, Konstan, Borchers, and Riedl, "An Algorithmic Framework for Performing Collaborative Filtering."

implementation would be somewhat more complex because it would require tracking the mean preference value for pairs of users.

In listing 4.3, passing the value `Weighting.WEIGHTED` to the constructor of `PearsonCorrelationSimilarity` as the second argument does this. It will cause the resulting correlation to be pushed towards 1.0, or –1.0, depending on how many data points were used to compute the correlation value. A quick rerun of the evaluation framework reveals that, in this case, this setting improves the score slightly to 0.77.

4.3.4 Defining similarity by Euclidean distance

Let's try `EuclideanDistanceSimilarity`—swap in this implementation by changing the `UserSimilarity` implementation used in listing 4.3 to `new EuclideanDistanceSimilarity(model)` instead.

This implementation is based on the *distance* between users. This idea makes sense if you think of users as points in a space of many dimensions (as many dimensions as there are items), whose coordinates are preference values. This similarity metric computes the Euclidean distance d between two such user points.[2] This value alone doesn't constitute a valid similarity metric, because larger values would mean more-distant, and therefore less similar, users. The value should be smaller when users are more similar. Therefore, the implementation actually returns $1 / (1+d)$. Refer to table 4.2, which illustrates this computation. You can verify that when the distance is 0 (users have identical preferences) the result is 1, decreasing to 0 as d increases. This similarity metric never returns a negative value, but larger values still mean more similarity.

Table 4.2 The Euclidean distance between user 1 and other users, and the resulting similarity scores

	Item 101	Item 102	Item 103	Distance	Similarity to user 1
User 1	5.0	3.0	2.5	0.000	1.000
User 2	2.0	2.5	5.0	3.937	0.203
User 3	2.5	-	-	2.500	0.286
User 4	5.0	-	3.0	0.500	0.667
User 5	4.0	3.0	2.0	1.118	0.472

After changing listing 4.3 to use `EuclideanDistanceSimilarity`, the result is 0.75—it happens to be a little better in this case, but barely. Note that some notion of similarity was computable for all pairs of users here, whereas the Pearson correlation couldn't produce an answer for users 1 and 3. This is good, on the one hand, though the result is based on one item in common, which could be construed as undesirable. This implementation also has the same possibly counterintuitive behavior: users 1 and 4 have a higher similarity than users 1 and 5.

[2] Recall that the Euclidian distance is the square root of the sum of the squares of the differences in coordinates.

4.3.5 Adapting the cosine measure similarity

The cosine measure similarity is another similarity metric that depends on envisioning user preferences as points in space. Hold in mind the image of user preferences as points in an *n*-dimensional space. Now imagine two lines from the origin, or point (0,0,...,0), to each of these two points. When two users are similar, they'll have similar ratings, and so will be relatively close in space—at least, they'll be in roughly the same direction from the origin. The angle formed between these two lines will be relatively small. In contrast, when the two users are dissimilar, their points will be distant, and likely in different directions from the origin, forming a wide angle.

This angle can be used as the basis for a similarity metric in the same way that the Euclidean distance was used to form a similarity metric. In this case, the cosine of the angle leads to a similarity value. If you're rusty on trigonometry, all you need to remember to understand this is that the cosine value is always between –1 and 1: the cosine of a small angle is near 1, and the cosine of a large angle near 180 degrees is close to –1. This is good, because small angles should map to high similarity, near 1, and large angles should map to near –1.

You may be searching for something like `CosineMeasureSimilarity` in Mahout, but you've already found it, under an unexpected name: `PearsonCorrelationSimilarity`. The cosine measure similarity and Pearson correlation aren't the same thing, but if you bother to work out the math, you'll discover that they actually reduce to the same computation when the two series of input values each have a mean of 0 (*centered*). The Mahout implementation centers the input, so the two are the same.

The cosine measure similarity is commonly referenced in research on collaborative filtering. You can employ this similarity metric by simply using `PearsonCorrelationSimilarity`.

4.3.6 Defining similarity by relative rank with the Spearman correlation

The Spearman correlation is an interesting variant on the Pearson correlation, for our purposes. Rather than compute a correlation based on the original preference values, it computes a correlation based on the *relative rank* of preference values. Imagine that, for each user, their least-preferred item's preference value is overwritten with a *1*. Then the next-least-preferred item's preference value is changed to *2*, and so on. To illustrate this, imagine that you were rating movies and gave your least-preferred movie one star, the next-least favorite two stars, and so on. Then, a Pearson correlation is computed on the transformed values. This is the Spearman correlation.

This process loses some information. Although it preserves the essence of the preference values—their ordering—it removes information about exactly how much more each item was liked than the last. This may or may not be a good idea; it's somewhere between keeping preference values and forgetting them entirely—two possibilities that we've explored before.

Table 4.3 shows the resulting Spearman correlations. Its simplifications on this already-simple data set result in some extreme values: in fact, all correlations are 1

Table 4.3 The preference values transformed into ranks, and the resulting Spearman correlation between user 1 and each of the other users

	Item 101	Item 102	Item 103	Correlation to user 1
User 1	3.0	2.0	1.0	1.0
User 2	1.0	2.0	3.0	–1.0
User 3	1.0	-	-	-
User 4	2.0	-	1.0	1.0
User 5	3.0	2.0	1.0	1.0

or –1 here, depending on whether a user's preference values run with or counter to user 1's preferences. As with the Pearson correlation, no value can be computed between users 1 and 3.

`SpearmanCorrelationSimilarity` implements this idea. You could try using this as the `UserSimilarity` in the evaluator code from before. Run it, and take a long coffee break. Turn in for the night. It won't finish anytime soon. This implementation is far slower because it must do some nontrivial work to compute and store these ranks. The Spearman correlation–based similarity metric is expensive to compute, and is therefore possibly more of academic interest than practical use. For some small data sets, though, it may be desirable.

This is a fine time to introduce one of many caching wrapper implementations available in Mahout. `CachingUserSimilarity` is a `UserSimilarity` implementation that wraps another `UserSimilarity` implementation and caches its results. That is, it delegates computation to another, given implementation, and remembers those results internally. Later, when it's asked for a user-user similarity value that was previously computed, it can answer immediately rather than ask the given implementation to compute it again. In this way, you can add caching to any similarity implementation. When the cost of performing a computation is relatively high, as here, it can be worthwhile to employ. The cost, of course, is the memory consumed by the cache.

Instead of just using `SpearmanCorrelationSimilarity`, try using the implementation shown in the following listing.

Listing 4.5 Employing caching with a `UserSimilarity` implementation

```
UserSimilarity similarity = new CachingUserSimilarity(
    new SpearmanCorrelationSimilarity(model), model);
```

It's also advisable to decrease the amount of test data from 5 percent to 1 percent by increasing the `trainingPercentage` argument to `evaluate()` from 0.95 to 0.99. It would also be wise to decrease the evaluation percentage from 5 percent to 1 percent by changing the last parameter from 0.05 to 0.01. This will allow the evaluation to finish in more like tens of minutes.

The result should be near 0.80. Again, broad conclusions are difficult to draw: on this particular data set, it was not quite as effective as other similarity metrics.

4.3.7 *Ignoring preference values in similarity with the Tanimoto coefficient*

Interestingly, there are also `UserSimilarity` implementations that ignore preference values entirely. They don't care whether a user expresses a high or low preference for an item—only that the user expresses a preference at all. As we discussed in chapter 3, it doesn't necessarily hurt to ignore preference values.

`TanimotoCoefficientSimilarity` is one such implementation, based on (surprise) the Tanimoto coefficient. This value is also known as the Jaccard coefficient. It's the number of items that two users express some preference for, divided by the number of items that either user expresses some preference for, as illustrated in figure 4.3.

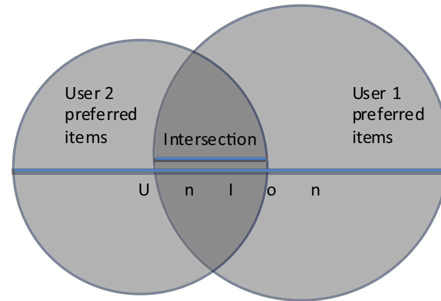


Figure 4.3 The Tanimoto coefficient is the ratio of the size of the intersection, or overlap, in two users' preferred items (the dark area), to the union of the users' preferred items (the dark and light areas together).

In other words, it's the ratio of the size of the intersection to the size of the union of their preferred items. It has the required properties: when two users' items completely overlap, the result is 1.0. When they have nothing in common, it's 0.0. The value is never negative, but that's OK. It's possible to expand the results into the range –1 to 1 with some simple math: $similarity = 2 \cdot similarity - 1$. It won't matter to the framework.

Table 4.4 shows the value of the Tanimoto coefficient similarity between user 1 and other users.

Note that this similarity metric doesn't depend only on the items that *both* users have some preference for, but that *either* user has some preference for. Hence, all seven items appear in the calculation, unlike before.

You're likely to use this similarity metric if, and only if, your underlying data contains only Boolean preferences and you have no preference values to begin with. If

Table 4.4 The similarity values between user 1 and other users, computed using the Tanimoto coefficient. Note that preference values themselves are omitted, because they aren't used in the computation.

	Item 101	Item 102	Item 103	Item 104	Item 105	Item 106	Item 107	Similarity to user 1
User 1	X	X	X					1.0
User 2	X	X	X	X				0.75
User 3	X			X	X		X	0.17
User 4	X		X	X		X		0.4
User 5	X	X	X	X	X	X		0.5

you do have preference values, you might use this metric because you believe there's more signal than noise in the values. You would usually do better with a metric that uses the preference information. In the GroupLens data set, using the Tanimoto coefficient gives a slightly worse score of 0.82.

4.3.8 Computing smarter similarity with a log-likelihood test

Log-likelihood–based similarity is similar to the Tanimoto coefficient–based similarity, though it's more difficult to understand intuitively. It's another metric that doesn't take account of individual preference values. The math involved in computing this similarity metric is beyond the scope of this book to explain. Like the Tanimoto coefficient, it's based on the number of items in common between two users, but its value is more an expression of how *unlikely* it is for two users to have so much overlap, given the total number of items out there and the number of items each user has a preference for.

To illustrate, consider two movie fans who have each seen and rated several movies but have only *both* seen *Star Wars* and *Casablanca*. Are they similar? If they've each seen hundreds of movies, it wouldn't mean much. Many people have seen these movies, and if these two have seen many movies but only managed to overlap in these two, they're probably not similar. On the other hand, if each user has seen just a few movies, and these two were on both users' lists, it would seem to imply that they're similar people when it comes to movies; the overlap would be significant.

The Tanimoto coefficient already encapsulates some of this thinking, because it looks at the ratio of the size of the intersection of their interests to the union. The log-likelihood is computing something slightly different. It's trying to assess how *unlikely* it is that the overlap between the two users is just due to chance. That is to say, two dissimilar users will no doubt happen to rate a couple movies in common, but two similar users will show an overlap that looks quite unlikely to be mere chance. With some statistical tests, this similarity metric attempts to determine just how strongly unlikely it is that two users have no resemblance in their tastes; the more unlikely, the more similar the two should be. The resulting similarity value may be interpreted as a probability that the overlap isn't due to chance.

These similarity values are shown for our small data set in table 4.5. As you can see, user 1's similarity with user 1, or any other user with identical preferred items, isn't 1.0 under this measure. Using a log-likelihood–based similarity metric is as easy as inserting `new LogLikelihoodSimilarity` in listing 4.3, as before.

Table 4.5 The similarity values between user 1 and other users, computed using the log-likelihood similarity metric

	Item 101	Item 102	Item 103	Item 104	Item 105	Item 106	Item 107	Similarity to user 1
User 1	X	X	X					0.90
User 2	X	X	X	X				0.84

Table 4.5 The similarity values between user 1 and other users, computed using the log-likelihood similarity metric *(continued)*

	Item 101	Item 102	Item 103	Item 104	Item 105	Item 106	Item 107	Similarity to user 1
User 3	X			X	X		X	0.55
User 4	X		X	X		X		0.16
User 5	X	X	X	X	X	X		0.55

Although it's hard to generalize, log-likelihood–based similarity will probably outperform Tanimoto coefficient–based similarity. It is, in a sense, a more intelligent metric. Rerunning the evaluation shows that, at least for this data set and recommender, it improves performance over `TanimotoCoefficientSimilarity`, to 0.73.

4.3.9 Inferring preferences

Sometimes *too little* data is a problem. In a few cases, for example, the Pearson correlation was unable to compute any similarity value at all because some pairs of users overlap in only one item. The Pearson correlation also can't take account of preference values for items for which only one user has expressed a preference.

What if a default value were filled in for all the missing data points? For example, the framework could pretend that each user has rated *every* item by *inferring* preferences for items for which the user hasn't explicitly expressed a preference. This sort of strategy is enabled via the `PreferenceInferrer` interface, which currently has one implementation, `AveragingPreferenceInferrer`. This implementation computes the average preference value for each user and fills in this average as the preference value for any item not already associated with the user. It can be enabled on a `UserSimilarity` implementation with a call to `setPreferenceInferrer()`.

Although this strategy is available, it's not usually helpful in practice. It's provided primarily because it's mentioned in early research papers on recommender engines. In theory, making up information purely based on existing information isn't adding anything, but it certainly does slow down computations drastically. It's available for experimentation but will likely not be useful when applied to real data sets.

You've now seen all of the user-user similarity metrics that are implemented within Mahout. This knowledge pays double dividends, because the very same implementations in Mahout also provide an analogous notion of item-item similarity. That is, the same computations can be applied to define how similar items are, not just users. This is just in time, because the notion of similarity is likewise the basis of the next style of Mahout recommender implementation you'll meet, the item-based recommender.

4.4 Item-based recommendation

We've looked at user-based recommenders in Mahout—not one recommender, but tools to build a nearly limitless number of variations on the basic user-based

approach, by plugging different and differently configured components into the implementation.

It's natural to look next at *item-based* recommenders. This section will be shorter, because several of the components in the preceding discussion (data models, similarity implementations) also apply to item-based recommenders.

Item-based recommendation is derived from how similar *items* are to *items*, instead of *users* to *users*. In Mahout, this means they're based on an `ItemSimilarity` implementation instead of `UserSimilarity`. To illustrate, let's return to the pair we left in the music store, doing their best to pick an album that a teenage boy would like. Imagine yet another line of reasoning they could have adopted:

ADULT: *I'm looking for a CD for a teenage boy.*

EMPLOYEE: *What kind of music or bands does he like?*

ADULT: *He wears a Bowling In Hades T-shirt all the time and seems to have all of their albums. Anything else you'd recommend?*

EMPLOYEE: *Well, about everyone I know that likes Bowling In Hades seems to like the new Rock Mobster album.*

This sounds reasonable. It's different from previous examples, too. The record store employee is recommending an item that's similar to something they already know the boy likes. This isn't the same as before, where the question was, "Who is similar to the boy, and what do they like?" Here the question is, "What is similar to what the boy likes?"

Figure 4.4 attempts to illustrate this difference, which is the essential difference between user-based and item-based recommenders. They go by different paths to find recommended items: via similar users, and via similar items, respectively.

4.4.1 The algorithm

The algorithm will feel familiar to you since you have seen user-based recommenders already. This is also how the algorithm is implemented in Mahout.

```
for every item i that u has no preference for yet
  for every item j that u has a preference for
    compute a similarity s between i and j
    add u's preference for j, weighted by s, to a running average
return the top items, ranked by weighted average
```

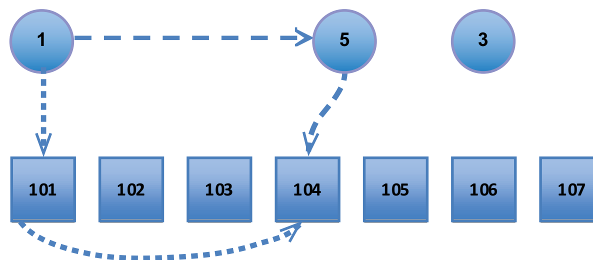


Figure 4.1 A basic illustration of the difference between user-based and item-based recommendation: user-based recommendation (large dashes) finds similar users, and sees what they like. Item-based recommendation (short dashes) sees what the user likes, and then finds similar items.

The third line shows how it's based on item-item similarities, not user-user similarities as before. The algorithms are similar, but not entirely symmetric. They do have notably different properties. For instance, the running time of an item-based recommender scales up as the number of items increases, whereas a user-based recommender's running time goes up as the number of users increases.

This suggests one reason that you might choose an item-based recommender: if the number of items is relatively low compared to the number of users, the performance advantage could be significant.

Also, items are typically less subject to change than users. When items are things like DVDs, it's reasonable to expect that over time, as you acquire more data, estimates of the similarities between items *converge*. There is no reason to expect them to change radically or frequently. Some of the same may be said of users, but users can change over time and new knowledge of users is likely to come in bursts of new information that must be digested quickly. To connect this to the last example, it's likely that Bowling In Hades albums and Rock Mobster albums will remain as similar to each other next year as today. But it's a lot less likely that the same fans mentioned above will have the same tastes next year, and so their similarities will change more.

If item-item similarities are more fixed, they're better candidates for precomputation. Precomputing similarities takes work, but it speeds up recommendations at runtime. This could be desirable in contexts where delivering recommendations quickly at runtime is essential—think about a news site that must potentially deliver recommendations immediately with each news article view.

In Mahout, the `GenericItemSimilarity` class can be used to precompute and store the results from any `ItemSimilarity`. It can be applied to any of the implementations you have seen, and can be added to the code snippets that follow, if you wish.

4.4.2 Exploring the item-based recommender

Let's insert a simple item-based recommender into our familiar evaluation framework, using the following code. It deploys a `GenericItemBasedRecommender` rather than `GenericUserBasedRecommender`, and it requires a different and simpler set of dependencies.

Listing 4.6 The core of a basic item-based recommender

```
public Recommender buildRecommender(DataModel model)
    throws TasteException {
  ItemSimilarity similarity = new PearsonCorrelationSimilarity(model);
  return new GenericItemBasedRecommender(model, similarity);
}
```

`PearsonCorrelationSimilarity` still works here because it also implements the `ItemSimilarity` interface, which is entirely analogous to the `UserSimilarity` interface. It implements the same notion of similarity, based on the Pearson correlation, but between items instead of users. That is, it compares series of preferences expressed by many users, for one item, rather than by one user for many items.

`GenericItemBasedRecommender` is simpler. It only needs a `DataModel` and `ItemSimilarity`—there's no `ItemNeighborhood`. You might wonder at the apparent asymmetry. Recall that the item-based recommendation process already begins with a limited number of starting points: the items that the user in question already expresses a preference for. This is analogous to the neighborhood of similar users that the user-based approach first identifies. It doesn't make sense in the second half of the algorithm to compute neighborhoods around each of the user's preferred items.

You're invited to experiment with different similarity metrics, as we did for users. Not all of the implementations of `UserSimilarity` also implement `ItemSimilarity`. You already know how to evaluate the accuracy of this item-based recommender when using various similarity metrics on the now-familiar GroupLens data set. The results are reproduced for convenience in table 4.6.

Implementation	Similarity
`PearsonCorrelationSimilarity`	0.75
`PearsonCorrelationSimilarity` + weighting	0.75
`EuclideanDistanceSimilarity`	0.76
`EuclideanDistanceSimilarity` + weighting	0.78
`TanimotoCoefficientSimilarity`	0.77
`LogLikelihoodSimilarity`	0.77

Table 4.6 Evaluation results under various `ItemSimilarity` metrics

One thing you may notice is that this recommender setup runs significantly faster. That's not surprising, given that the data set has about 70,000 users and 10,000 items. Item-based recommenders are generally faster when there are fewer items than users. You may, as a result, wish to increase the percentage of data used in the evaluation to 20 percent or so (pass 0.2 as the final argument to `evaluate()`). This should result in a more reliable evaluation. Note that there's little apparent difference among these implementations on this data set.

4.5 Slope-one recommender

Did you like the movie *Carlito's Way*? Most people who liked this movie, it seems, also liked another film starring Al Pacino, *Scarface*. But people tend to like *Scarface* a bit more. We'd imagine most people who think of *Carlito's Way* as a four-star movie would give *Scarface* five stars. So if you told us you thought *Carlito's Way* was a three-star movie, we might guess you'd give *Scarface* four stars—one more than the other film.

If you agree with this sort of reasoning, you'll like the slope-one recommender (http://en.wikipedia.org/wiki/Slope_One). It estimates preferences for new items based on average difference in the preference value (*diffs*) between a new item and the other items the user prefers.

For example, let's say that, on average, people rate *Scarface* higher by 1.0 than *Carlito's Way*. Let's also say that everyone rates *Scarface* the same as *The Godfather*, on average. And now, there's a user who rates *Carlito's Way* as 2.0, and *The Godfather* as 4.0. What's a good estimate of their preference for *Scarface*?

Based on *Carlito's Way*, a good guess would be 2.0 + 1.0 = 3.0. Based on *The Godfather*, it might be 4.0 + 0.0 = 4.0. A better guess still might be the average of the two: 3.5. This is the essence of the slope-one recommender approach.

4.5.1 The algorithm

The slope-one name comes from the fact that the recommender algorithm starts with the assumption that there's some linear relationship between the preference values for one item and another, that it's valid to estimate the preferences for some item Y based on the preferences for item X, via some linear function like $Y = mX + b$. Then the slope-one recommender makes the additional simplifying assumption that m=1: slope one. It just remains to find $b = Y-X$, the (average) difference in preference value, for every pair of items.

This means the algorithm consists of a significant preprocessing phase, in which all item-item preference value differences are computed:

```
for every item i
  for every other item j
    for every user u expressing preference for both i and j
      add the difference in u's preference for i and j to an average
```

And then, the recommendation algorithm looks like this:

```
for every item i the user u expresses no preference for
  for every item j that user u expresses a preference for
    find the average preference difference between j and i
    add this diff to u's preference value for j
    add this to a running average
return the top items, ranked by these averages
```

The average diffs over the small sample recommender inputs from several prior examples in the book are shown in table 4.7.

Table 4.7 Average differences in preference values between all pairs of items. Cells along the diagonal are 0.0. Cells in the bottom left are simply the negative of their counterparts across the diagonal, so these aren't represented explicitly. Some diffs don't exist, such as 102-107, because no user expressed a preference for both 102 and 107.

	Item 101	Item 102	Item 103	Item 104	Item 105	Item 106	Item 107
Item 101		–0.833	0.875	0.25	0.75	–0.5	2.5
Item 102			0.333	0.25	0.5	1.0	-
Item 103				0.167	1.5	1.5	-
Item 104					0.0	–0.25	1.0

	Item 101	Item 102	Item 103	Item 104	Item 105	Item 106	Item 107
Item 105						0.5	0.5
Item 106							-
Item 107							

Slope-one is attractive because the online portion of the algorithm is fast. Like an item-based recommender, its performance doesn't depend upon the number of users in the data model. It depends only upon the average preference difference between every pair of items, which can be precomputed. Further, its underlying data structure can be efficiently updated: when a preference changes, it's simple to update the relevant diff values. In contexts where preferences may change quickly, this is an asset.

Note that the memory requirements necessary to store all of these item-item differences in preference values grow as the *square* of the number of items. Twice as many items means four times the memory!

4.5.2 Slope-one in practice

You can easily try the slope-one recommender. It takes no similarity metric as a necessary argument:

```
new SlopeOneRecommender(model)
```

After running a standard evaluation using, again, the GroupLens 10 million rating data set, you'll get a result near 0.65. That's the best yet. Indeed, the simple slope-one approach works well in many cases. This algorithm doesn't make use of a similarity metric, unlike the other approaches so far. It has relatively few knobs to twiddle.

Like the Pearson correlation, the simplest form of the slope-one algorithm has a vulnerability: item-item diffs are given equal weighting regardless of how reliable they are—how much data they're based upon. Let's say only one user in the history of movie watching has rated both *Carlito's Way* and *The Notebook*. It's possible; they're quite different films. It's easy to compute a diff for these two films, but would it be as useful as the diff between *Carlito's Way* and *The Godfather*, which are averaged over thousands of users? It sounds unlikely. The latter diff is probably more reliable because it's an average over a higher number of users.

Again, some form of weighting may help to improve these recommendations. `SlopeOneRecommender` offers two types of weighting: weighting based on count and on standard deviation. Recall that slope-one estimates the preference values by adding diffs to all of the user's current preference values and then averaging all of those results together to form an estimate. Count weighting will weight more heavily those diffs that are based on more data. The average becomes a weighted average, where the diff count is the weight—the number of users on which the diff is based.

Similarly, standard deviation weighting will weight according to the standard deviation of difference in preference value. A lower standard deviation means a higher

weighting. If the difference in preference value between two films is very consistent across many users, it seems more reliable and should be given more weight. If it varies considerably from user to user, it should be deemphasized.

These variants turn out to be enough of a good idea that they're enabled by default. You already used this strategy when you ran the previous evaluation. Disable them to see the effect, as follows.

Listing 4.7 Selecting no weighting with a `SlopeOneRecommender`

```
DiffStorage diffStorage = new MemoryDiffStorage(
    model, Weighting.UNWEIGHTED, Long.MAX_VALUE));
return new SlopeOneRecommender(
 model,
 Weighting.UNWEIGHTED,
 Weighting.UNWEIGHTED,
 diffStorage);
```

The result is 0.67—only slightly worse on this data set.

4.5.3 DiffStorage and memory considerations

Slope-one does have its price: memory consumption. In fact, if you tweak the evaluation to use just 10 percent of all data (about 100,000 ratings), even a gigabyte of heap space won't be enough. The diffs are used so frequently, and it's so relatively expensive to compute them, that they do need to be computed and stored ahead of time. But keeping them all in memory can get expensive. It may become necessary to store diffs elsewhere.

Fortunately, implementations like `MySQLJDBCDiffStorage` exist for this purpose, allowing diffs to be computed and updated from a database. It must be used in conjunction with a JDBC-backed `DataModel` implementation like `MySQLJDBCDataModel`, as shown in the next listing.

Listing 4.8 Creating a JDBC-backed `DiffStorage`

```
AbstractJDBCDataModel model = new MySQLJDBCDataModel();
DiffStorage diffStorage = new MySQLJDBCDiffStorage(model);
Recommender recommender = new SlopeOneRecommender(
    model, Weighting.WEIGHTED, Weighting.WEIGHTED, diffStorage);
```

As with `MySQLJDBCDataModel`, the table name and column names used by `MySQLJDBC-DiffStorage` can be customized via constructor parameters.

4.5.4 Distributing the precomputation

Precomputing the item-item diffs is significant work. Although it's more likely that the size of your data will cause problems with memory requirements before the time required to compute these diffs becomes problematic, you might be wondering if there are ways to distribute this computation to complete it faster. Diffs can be

updated easily at runtime in response to new information, so a relatively infrequent offline precomputation process is feasible in this model.

Distributing the diff computation via Hadoop *is* supported. Chapter 6 will introduce all of the Hadoop-related recommender support in Mahout, to explore this process.

4.6 New and experimental recommenders

Mahout also contains implementations of other approaches to recommendation. The three implementations presented briefly in this section are newer: the implementations may still be evolving, or the technique may be more recent and experimental. All are worthy ideas that may be useful for use or for modification.

4.6.1 Singular value decomposition–based recommenders

Among the most intriguing of these implementations is `SVDRecommender`, based on the singular value decomposition (SVD). This is an important technique in linear algebra that pops up in machine learning techniques. Fully understanding it requires some advanced matrix math and understanding of matrix factorization, but this isn't necessary to appreciate SVD's application to recommenders.

To outline the intuition behind what SVD does for recommenders, let's say you ask a friend what sort of music she likes, and she lists the following artists:

- Brahms
- Chopin
- Miles Davis
- Tchaikovsky
- Louis Armstrong
- Schumann
- John Coltrane
- Charlie Parker

She might as well have summarized that she likes classical and jazz music. That communicates less precise information, but not a great deal less. From either statement, you could (probably correctly) infer that she would appreciate Beethoven more than the classic rock band Deep Purple.

Of course, recommender engines operate in a world of many specific data points, not generalities. The input is user preferences for a lot of particular items—more like the preceding list than this summary. It would be nice to operate on a smaller set of data, all else equal, for reasons of performance. If, for example, iTunes could base its Genius recommendations not on *billions* of individual song ratings, but instead on *millions* of ratings of genres, it would be faster—and, as a basis for recommending music, it might not be much worse.

Here, SVD is the magic that can do the equivalent of the preceding summarization. It boils down the world of user preferences for individual items to a world of user preferences for more general and less numerous *features* (like genre). This is, potentially, a much smaller set of data.

Although this process loses some information, it can sometimes improve recommendation results. The process smooths the input in useful ways. For example, imagine two car enthusiasts. One loves Corvettes, and the other loves Camaros, and they

want car recommendations. These enthusiasts have similar tastes: both love a Chevrolet sports car. But in a typical data model for this problem, these two cars would be different items. Without any overlap in their preferences, these two users would be deemed unrelated. However, an SVD-based recommender would perhaps find the similarity. The SVD output may contain features that correspond to concepts like *Chevrolet* or *sports car*, with which both users would be associated. From the overlap in features, a similarity could be computed.

Using the `SVDRecommender` is as simple as this:

```
new SVDRecommender(model, new ALSWRFactorizer(model, 10, 0.05, 10))
```

`SVDRecommender` uses a `Factorizer` to perform its work; a fine first choice is the `ALSWRFactorizer` implementation used here. We won't discuss your choices of `Factorizer` in detail here.

The first numeric argument is the number of features that the SVD should target. There's no right answer for this; it would be equivalent to the number of genres that you condensed someone's musical taste into, in this section's first example. The second argument is lambda and it controls a feature of the factorizer called regularization. The last argument is the number of training steps to run. Think of this as controlling the amount of time it should spend producing this summary; larger values mean longer training.

This approach can give good results (0.69 on the GroupLens data set). At the moment, the major issue with the implementation is that it computes the SVD in memory. This requires the entire data set to fit in memory, and it's precisely when this *isn't* the case that this technique is appealing, because it can shrink the input without compromising output quality significantly. In the future, this algorithm will be reimplemented in terms of Hadoop, so the necessarily massive SVD computation can be distributed across multiple machines. It isn't yet available at this stage of Mahout's evolution.

4.6.2 Linear interpolation item–based recommendation

Linear interpolation is a somewhat different take on item-based recommendation, implemented as `KnnItemBasedRecommender`. *Knn* is short for *k nearest neighbors*, which is an idea also embodied in `NearestNUserNeighborhood`. As you saw earlier in figure 4.1, a `UserNeighborhood` implementation selects a fixed number of most similar users as a neighborhood of similar users. This linear interpolation algorithm uses the concept of a user neighborhood, but in a different way.

`KnnItemBasedRecommender` still estimates preference values by means of a weighted average of the items the user already has a preference for, but the weights aren't the results of a similarity metric. Instead, the algorithm calculates the optimal set of weights to use between all pairs of items by means of some linear algebra—here's where the linear interpolation comes in. Yes, it's possible to optimize the weights with some mathematical wizardry.

In reality, it would be very expensive to compute this across all pairs of items, so instead it first calculates a neighborhood of items most similar to the target item—the item for which a preference is being estimated. It chooses the n nearest neighbors, in much the same way that `NearestNUserNeighborhood` did. One can try `KnnItemBased-Recommender` as shown in the following listing.

Listing 4.9 Deploying `KnnItemBasedRecommender`

```
ItemSimilarity similarity = new LogLikelihoodSimilarity(model);
Optimizer optimizer = new NonNegativeQuadraticOptimizer();
return new KnnItemBasedRecommender(model, similarity, optimizer, 10);
```

This code will cause the recommender to use a log-likelihood similarity metric to calculate nearest-10 neighborhoods of items. Then it'll use a quadratic programming–based strategy to calculate the linear interpolation. The details of this calculation are outside the scope of the book.

The implementation is quite functional, but in its current form, it's also slow on moderately sized data sets. It should be viewed as viable for small data sets, or for study and extension. On the GroupLens data set, it yields an evaluation result of 0.76.

4.6.3 *Cluster-based recommendation*

Cluster-based recommendation is best thought of as a variant on user-based recommendation. Instead of recommending items to users, items are recommended to clusters of similar users. This entails a preprocessing phase, in which all users are partitioned into clusters. Recommendations are then produced for each cluster, such that the recommended items are most interesting to the largest number of users.

The upside of this approach is that recommendation is fast at runtime because almost everything is precomputed. One could argue that the recommendations are less personal this way, because recommendations are computed for a group rather than an individual. This approach may be more effective at producing recommendations for new users, who have little preference data available. As long as the user can be attached to a reasonably relevant cluster, the recommendations ought to be as good as they'll be when more is known about the user.

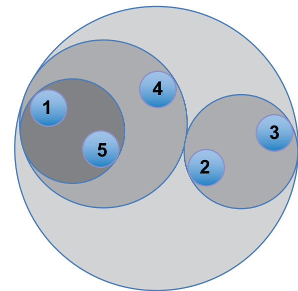


Figure 4.5 An illustration of clustering. Users 1 and 5 are clustered together first, as are 2 and 3, because they're closest. Then, 4 is clustered with the 1-5 cluster to create a larger cluster, one step up in the "tree."

The name comes from the fact that the algorithm repeatedly joins most-similar clusters into larger clusters, and this implicitly organizes users into a sort of hierarchy, or tree, as depicted in figure 4.5.

Unfortunately, the clustering takes a long time, as you'll see if you attempt to run the code in the following listing, which employs a `TreeClusteringRecommender` to implement this idea.

Listing 4.10 Creating a cluster-based recommender

```
UserSimilarity similarity = new LogLikelihoodSimilarity(model);
ClusterSimilarity clusterSimilarity =
    new FarthestNeighborClusterSimilarity(similarity);
return new TreeClusteringRecommender(model, clusterSimilarity, 10);
```

Similarity between users is, as usual, defined by a `UserSimilarity` implementation. Similarity between two clusters of users is defined by a `ClusterSimilarity` implementation. Currently, two implementations are available: one defines cluster similarity as the similarity between the two *most similar* user pairs, one chosen from each cluster; the other defines cluster similarity as the similarity between the two *least similar* users.

Both approaches are reasonable; in both cases the risk is that one outlier on the edge of a cluster distorts the notion of cluster similarity. Two clusters whose members are on average distant but happen to be close at one edge would be considered quite close by the most-similar-user rule, which is implemented by `NearestNeighborClusterSimilarity`. The least-similar-user rule, implemented by `FarthestNeighborClusterSimilarity` in listing 4.10, likewise may consider two fairly close clusters to be distant from one another if each contains an outlier far away from the opposite cluster.

A third approach, defining cluster similarity as the distance between the centers, or means, of two clusters, is also possible, though not yet implemented in this part of Mahout.

4.7 Comparison to other recommenders

As mentioned earlier in the book, content-based recommendation is a broad and often-mentioned approach to recommendation that takes into account the *content* or attributes of items. For this reason, it's similar to, yet distinct from, collaborative filtering approaches, which are based on user associations with items only, and treats items as black boxes without attributes. Although Mahout largely doesn't implement content-based approaches, it does offer some opportunities to make use of item attributes in recommendation computations.

4.7.1 Injecting content-based techniques into Mahout

For example, consider an online bookseller who stocks multiple editions of some books. This seller might recommend books to its customers. Its items are books, of course, and it might naturally define a book by its ISBN number (a unique product identifier). But for a popular public-domain book like *Jane Eyre*, there may be many printings by different publishers of the same text, under different ISBN numbers. It seems more natural to recommend books based on their texts, rather than on a particular edition—do you care more about reading "*Jane Eyre*" or "*Jane Eyre* as printed by ACME Publications in 1993 in paperback?" Rather than treat various publications of *Jane Eyre* as distinct items, it might be more useful to think of the book itself, the text, as the item, and recommend all editions of this book equally. This would be, in a sense, content-based recommendation. By treating the underlying text of a book

product, which is its dominant *attribute*, as the *item* in a collaborative filtering sense, and then applying collaborative filtering techniques with Mahout, the seller would be engaging in a form of content-based recommendation.

Or recall that item-based recommenders require some notion of similarity between two given items. This similarity is encapsulated by an `ItemSimilarity` implementation. So far, implementations have derived similarity from user preferences only—this is classic collaborative filtering. But there's no reason the implementation couldn't be based on item attributes. For example, a movie recommender might define item (movie) similarity as a function of movie attributes like genre, director, actor, and year of release. Using such an implementation within a traditional item-based recommender would also be an example of content-based recommendation.

4.7.2 Looking deeper into content-based recommendation

Taking this a step further, imagine content-based recommendation as a generalization of collaborative filtering. In collaborative filtering, computations are based on preferences, which are user-item associations. But what drives these user-item associations? It's likely that users have implicit preferences for certain item *attributes*, which come out in their preferences for certain items and not others. For example, if your friend told you she likes the albums *Led Zeppelin I*, *Led Zeppelin II*, and *Led Zeppelin III*, you might well guess she is actually expressing a preference for an attribute of these items: the band Led Zeppelin. By discovering these associations, and discovering attributes of items, it's possible to construct recommender engines based on these more nuanced understandings of user-item associations.

These techniques come to resemble search and document-retrieval techniques: asking what items a user might like based on user-attribute associations and item attributes resembles retrieving search results based on query terms and the occurrence of terms in documents. Although Mahout's recommender support doesn't yet embrace these techniques, it's a natural direction for future versions to address.

4.8 Comparison to model-based recommenders

Another future direction for Mahout is model-based recommendation. This family of techniques attempts to build some model of user preferences based on existing preferences, and then infer new preferences. These techniques generally fall into the broader category of collaborative filtering, because they typically derive from user preferences only.

The model might be a probabilistic picture of users' preferences, in the form of a Bayesian network, for example. The algorithm then attempts to judge the *probability* of liking an item given its knowledge of all user preferences, and it ranks the recommendations accordingly.

Association rule learning can be applied in a similar sense to recommendations. By learning rules such as "when a user prefers item X and item Y, they will prefer item Z" from the data, and judging confidence in the reliability of such rules, a recommender can put together the most likely set of new, preferred items.

Cluster-based recommenders might be considered a type of model-based recommender. The clusters represent a model of how users group together and therefore how their preferences might run the same way. In this limited sense, Mahout supports model-based recommenders. But this is an area that's still largely under construction in Mahout as of this writing.

4.9 Summary

In this chapter, we thoroughly explored the core recommender algorithms offered by Mahout.

We started by explaining the general user-based recommender algorithm in terms of real-world reasoning. From there, we looked at how this algorithm is realized in Mahout, as `GenericUserBasedRecommender`. Many pieces of this generic approach can be customized, such as the definition of user similarity and user neighborhood.

We looked at the classic user similarity metric, based on the Pearson correlation, noted some possible issues with this approach, and looked at responses such as weighting. We looked at similarity metrics based on the Euclidean distance, Spearman correlation, Tanimoto coefficient, and log-likelihood ratio.

Then we examined another canonical recommendation technique, item-based recommendation, as implemented by `GenericItemBasedRecommender`. It reuses some concepts already covered in the context of user-based recommenders, such as the Pearson correlation.

Next, we examined a slope-one recommender, a unique and relatively simple approach to recommendation based on average differences in preference values between items. It requires significant precomputation and storage for these diffs, so we explored how to store these both in memory and in a database.

Last, we looked briefly at a few newer, more experimental implementations currently in the framework. These include implementations based on singular value decomposition, linear interpolation, and clustering. These may be useful for small data sets, or may be of academic interest, because they're still a work in progress.

The key parameters and features for each implementation are summarized in table 4.8.

Table 4.8 Summary of available recommender implementations in Mahout, their key input parameters, and key features to consider when choosing an implementation

Implementation	Key parameters	Key features
`GenericUserBasedRecommender`	• User similarity metric • Neighborhood definition and size	• Conventional implementation • Fast when number of users is relatively small
`GenericItemBasedRecommender`	• Item similarity metric	• Fast when number of items is relatively small • Useful when an external notion of item similarity is available

Table 4.8 Summary of available recommender implementations in Mahout, their key input parameters, and key features to consider when choosing an implementation *(continued)*

Implementation	Key parameters	Key features
`SlopeOneRecommender`	• Diff storage strategy	• Recommendations and updates are fast at runtime • Requires large precomputation • Suitable when number of items is relatively small
`SVDRecommender`	• Number of features	• Good results • Requires large precomputation
`KnnItemBasedRecommender`	• Number of means (k) • Item similarity metric • Neighborhood size	• Good when number of items is relatively small
`TreeClusteringRecommender`	• Number of clusters • Cluster similarity definition • User similarity metric	• Recommendations are fast at runtime • Requires large precomputation • Good when number of users is relatively small

We're done introducing Mahout's recommender engine support. Now we're ready to examine even larger and more realistic data sets, from the practitioner's perspective. You might wonder why we've made little mention of Hadoop yet. Hadoop is a powerful tool, and it's necessary when dealing with massive data sets, where one must make use of many machines. This has drawbacks: such computations are massive, they're resource-intensive, and they complete in hours, not milliseconds. We'll reach Hadoop in the last chapter in this section.

First, in the next chapter, we create a production-ready recommender engine based on Mahout that fits onto one machine—one that can respond to requests for recommendations in a fraction of a second and incorporate updates immediately.

5 Taking recommenders to production

This chapter covers

- Analyzing data from a real dating site
- Designing and refining a recommender engine solution
- Deploying a web-based recommender service in production

So far, this book has toured the recommender algorithms and variants that Apache Mahout provides, and discussed how to evaluate the accuracy and performance of a recommender. The next step is to apply all of this to a real data set to create an effective recommender engine from scratch based on data. You'll create one based on data taken from a dating site, and then you'll turn it into a deployable, production-ready web service.

There's no one standard approach to building a recommender for given data and a given problem domain. The data must at least represent associations between users and items—where users and items might be many things. Adapting the input to recommender algorithms is usually quite a problem-specific process. How you

discover the *best* recommender engine to apply to the input data is likewise specific to each context. It inevitably involves hands-on exploration, experimentation, and evaluation on real problem data.

This chapter presents one end-to-end example that suggests the process you might follow to develop a recommender system using Mahout for your data set. You'll try an approach, collect data, evaluate the results, and repeat many times. Many approaches won't go anywhere, but that's good information as well. This brute force approach is appropriate, because it's relatively painless to evaluate an approach in Mahout, and because here, as in other problem domains, it's not at all clear what the right approach is from just looking at the data.

5.1 Analyzing example data from a dating site

This example will use a new data set, derived from the Czech dating site Líbímseti (http://libimseti.cz/). Users of this site are able to rate other users' profiles on a scale of 1 to 10. A 1 means *NELÍBÍ*, or *dislike*, and a 10 means *LÍBÍ*, or *like*. The presentation of profiles on the site suggests that users of such a site are expressing some assessment of the profiled user's appeal, attractiveness, and dateability. A great deal of this data has been anonymized, made available for research, and published by Vaclav Petricek (http://www.occamslab.com/petricek/data/).[1] You'll need to download a copy of the complete data from this site, at http://www.occamslab.com/petricek/data/libimseti-complete.zip, to use in this chapter's examples.[2]

With 17,359,346 ratings in the data set, this is almost twice as big as our previous GroupLens data set. It contains users' explicit ratings for items, where items are other people's user profiles. That means a recommender system built on this data will be recommending *people* to people. This is a reminder to think broadly about recommenders, which aren't limited to recommending objects like books and DVDs.

The first step in creating a recommender is analyzing what data you have to work with, and beginning to form ideas about which recommender algorithm might be suitable to use with it. The 257 MB ratings.dat file in the archive you downloaded is a simple comma-delimited file containing user ID, profile ID, and rating. Each line represents one user's rating of one other user's profile. The data is purposely obfuscated, so the user IDs aren't real user IDs from the site. Profiles are *user* profiles, so this data represents users' ratings of other users. One might suppose that user IDs and profile IDs are comparable here, that user ID 1 and profile ID 1 are the same user, but this doesn't appear to be the case, likely for reasons of anonymity.

There are 135,359 unique users in the data, who together rated 168,791 unique user profiles. Because the number of users and items are about the same, neither user-based nor item-based recommendation is obviously more efficient. If there had been a great deal more profiles than users, an item-based recommender would have been

[1] See also Lukas Brozovsky and Vaclav Petricek, "Recommender System for Online Dating Service,"http://www.occamslab.com/petricek/papers/dating/brozovsky07recommender.pdf

[2] Neither the site nor the publisher of the data endorse or are connected with this book.

slower. Slope-one can be applied here, even though its memory requirements scale up quickly with the number of items; its memory requirements can be capped.

The data set has been preprocessed in a way: no users who produced less than 20 ratings are included. In addition, users who seem to have rated every profile with the same value are excluded, presumably because it may be spam or an unserious attempt at rating. The data that's given comes from users who bothered to make a number of ratings; presumably, their input is useful, and not noisy, compared to the ratings of less-engaged users.

This input is already formatted for use with Mahout's `FileDataModel`. The user and profile IDs are numeric, and the file is already comma-delimited with fields in the required order: user ID, item ID, preference value.

The data set provides another interesting set of data: the gender of the user for many of the profiles in the data set. The gender isn't given for all profiles; in gender.dat, several lines end in *U* which means *unknown*. The gender of the users isn't given in the data set—just gender of the profiles—but that means we know something more about each item. When being recommended, male profiles are more like each other than they are like female profiles, and vice versa. If most or all of a user's ratings are for male profiles, it stands to reason that the user will rate male profiles as far more desirable dates than female. This information could be the basis for an item-item similarity metric.

This isn't a perfect assumption. Without becoming sidetracked on sensitive issues of sexuality, note that some users of the site may enjoy rating profiles of a gender they aren't interested in dating, for fun. Some users may also legitimately have some romantic interest in both genders. In fact, the very first two ratings in ratings.dat are from one user, and yet appear to be for profiles of different genders.

It's important to account for gender in a dating site recommender engine like this; it would be bad to recommend a female to a user interested only in males—this would surely be viewed as a bad recommendation, and, to some, offensive. This restriction is important, but it doesn't fit neatly into the standard recommender algorithms we've seen in Mahout. Later sections in this chapter will examine how to inject this information as both a filter and a similarity metric.

5.2 Finding an effective recommender

To create a recommender engine for the Líbímseti data, you need to pick a recommender implementation from Mahout. The recommender ought to be both fast and produce good recommendations. Of those two, it's better to focus on producing good recommendations first, and then look to performance. After all, what's the use in producing bad answers quickly?

It's impossible to deduce the right implementation from looking at the data; some empirical testing is needed. Armed with the recommender evaluation framework from chapter 2, the next step is to collect some information on how well various implementations perform on this data set.

5.2.1 User-based recommenders

User-based recommenders are a natural first stop. Several different similarity metrics and neighborhood definitions are available in Mahout. To get some sense of what works and what doesn't, you can try many combinations. The result of some such experimenting in our test environment is summarized in tables 5.1 and 5.2, and figures 5.1 and 5.2.

Table 5.1 Average absolute difference in estimated and actual preferences when evaluating a user-based recommender using one of several similarity metrics, and using a nearest-n user neighborhood

Similarity	n = 1	n = 2	n = 4	n = 8	n = 16	n = 32	n = 64	n = 128
Euclidean	1.17	**1.12**	1.23	1.25	1.25	1.33	1.48	1.43
Pearson	1.30	1.19	1.27	1.30	1.26	1.35	1.38	1.47
Log-likelihood	1.33	1.38	1.33	1.35	1.33	1.29	1.33	1.49
Tanimoto	1.32	1.33	1.43	1.32	1.30	1.39	1.37	1.41

Table 5.2 Average absolute difference in estimated and actual preferences when evaluating a user-based recommender using one of several similarity metrics, and using a threshold-based user neighborhood. Some values are "not a number," or undefined, and are denoted by Java's NaN symbol.

Similarity	t = 0.95	t = 0.9	t = 0.85	t = 0.8	t = 0.75	t = 0.7
Euclidean	**1.33**	1.37	1.39	1.43	1.41	1.47
Pearson	1.47	1.4	1.42	1.4	1.38	1.37
Log-likelihood	1.37	1.46	1.56	1.52	1.51	1.43
Tanimoto	*NaN*	*NaN*	*NaN*	*NaN*	*NaN*	*NaN*

These scores aren't bad. These recommenders are estimating user preferences to within 1.12 to 1.56 points on a scale of 1 to 10, on average.

There are some trends here, even though some individual evaluation results vary from that trend. It looks like the Euclidean distance similarity metric may be a little better than Pearson, though their results are quite similar. It also appears that using a

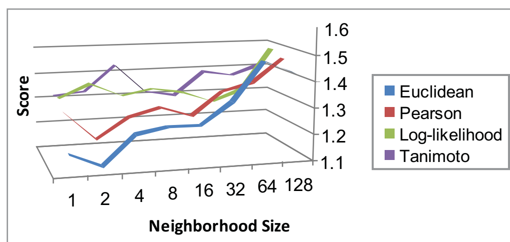


Figure 5.1 Visualization of values in table 5.1

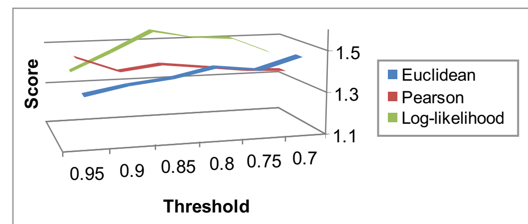


Figure 5.2 Visualization of values in table 5.2

small neighborhood is better than a large one; the best evaluations occur when using a neighborhood of two people! Maybe users' preferences are truly quite personal, and incorporating too many others in the computation doesn't help.

What explains the NaN (not a number) result for the Tanimoto coefficient–based similarity metric? It's listed here to highlight a subtle point about this methodology. Although all similarity metrics return a value between –1 and 1 and return higher values to indicate greater similarity, it's not true that any given value means the same thing for each similarity metric. This is generally true, and not an artifact of how Mahout works. For example, 0.5 from a Pearson correlation-based metric indicates moderate similarity. A value of 0.5 for the Tanimoto coefficient indicates significant similarity between two users: of all items known to either of them, half are known to both.

Even though thresholds of 0.7 to 0.95 were reasonable values to test for the other metrics, these are quite high for a Tanimoto coefficient–based similarity metric. In each case, the bar was set so high that no user neighborhood was established in any test case! Here, it might have been more useful to test thresholds from, say, 0.4 on down. In fact, with a threshold of 0.3, the best evaluation score approaches 1.2.

Similarly, although there's an apparent best value for *n* in the nearest-*n* user neighborhood data, there isn't quite the same best value in the threshold-based user neighborhood results. For example, the Euclidean distance–based similarity metric seems to be producing better results as the threshold increases. Perhaps the most valuable users to include in the neighborhood have a Euclidean-based similarity of over 0.95. What happens at 0.99? Or 0.999? The evaluation result goes down to about 1.35; not bad, but not apparently the best recommender.

You can continue looking for even better configurations. For our purposes here, though, we take the current best solution in Mahout to be

- User-based recommender
- Euclidean distance similarity metric
- Nearest-2 neighborhood

5.2.2 Item-based recommenders

Item-based recommenders involve just one choice: an item similarity metric. Trying each similarity metric is a straightforward way to see what works best. Table 5.3 summarizes the outcomes.

Table 5.3 Average absolute differences in estimated and actual preferences, when evaluating an item-based recommender using several different similarity metrics

Similarity	Score
Euclidean	2.36
Pearson	2.32
Log-likelihood	2.38
Tanimoto	2.40

Scores are notably worse here; the average error, or difference between estimated and actual preference values, has roughly doubled to over 2. For this data, the item-based approach isn't as effective, for some reason. Why? Before, the algorithm computed

similarities between users in a user-based approach, based on how users rated other users' profiles. Now, it's computing the similarity between user profiles based on how other users rated that profile. Maybe this isn't as meaningful—maybe ratings say more about the *rater* than the *rated profile*.

Whatever the explanation, it seems clear from these results that item-based recommendation isn't the best choice here.

5.2.3 Slope-one recommender

Recall that the slope-one recommender constructs a diff for most item-item pairs in the data model. With 168,791 items (profiles) here, this means storing potentially 28 *billion* diffs—far too much to fit in memory. Storing these diffs in a database is possible, but it'll greatly slow performance.

Fortunately, there is another option, which is to ask the framework to limit the number of diffs stored to perhaps ten million, as shown in listing 5.1. The framework will attempt to choose the most useful diffs to keep. Most useful here means those diffs between a pair of items that turn up most often together in the list of items associated with a user. For example, if items A and B appear in the preferences of hundreds of users, the average diff in their preference values is likely significant, and useful. If A and B only appear together in the preferences of one user, it's more like a fluke than a piece of data worth storing.

Listing 5.1 Limiting memory consumed by `MemoryDiffStorage`

```
DiffStorage diffStorage = new MemoryDiffStorage(
    model, Weighting.WEIGHTED, 10000000L);
return new SlopeOneRecommender(
    model, Weighting.WEIGHTED, Weighting.WEIGHTED, diffStorage);
```

From examining the Mahout log output, this approach keeps memory consumption to about 1.5 GB. You'll also notice again how fast slope-one is; on the workstation we used for testing, the average recommendation times were under 10 milliseconds, compared to 200 milliseconds or so for other algorithms.

The evaluation result is about 1.41. This isn't a bad result, but not quite as good as observed with the user-based recommenders. It's likely not worth pursuing slope-one for this particular data set.

5.2.4 Evaluating precision and recall


No. 5

The preceding examples experimented with a Tanimoto coefficient–based similarity metric and a log-likelihood–based metric, neither of which use preference values. But they didn't evaluate recommenders that completely ignore rating values. Such recommenders can't be evaluated in the same way—there are no estimated preference values to compare against real values because there are no preference values at all.

It's possible to examine the precision and recall of such recommenders against the current best solution (user-based recommender with a Euclidean distance metric and

a nearest-2 neighborhood) by using a `RecommenderIRStatsEvaluator`. This evaluator reveals that precision and recall at 10 (that is, precision and recall as judged by looking at the first 10 recommendations) are about 3.6 and 5 percent, respectively. This seems low: the recommender rarely recommends the users' own top-rated profiles when those top-rated profiles are removed. In this context, that's not obviously a bad thing. It's conceivable that a user might see plenty of perfect 10s on such a dating site and perhaps has only ever encountered and rated some of them. It could be that the recommender is suggesting even more desirable profiles than the user has seen! Certainly, this is what the recommender is communicating, that the users' top-rated profiles aren't usually the ones they would like most, were they to actually review every profile in existence.

The other explanation, of course, is that the recommender isn't functioning well. But this recommender is fairly good at estimating preference values, usually estimating ratings within about 1 point on a 10-point scale. So the explanation in the previous paragraph could be valid.

An interesting thing happens when the rating data is ignored by using Mahout's `GenericBooleanPrefDataModel`, `GenericBooleanPrefUserBasedRecommender`, and an appropriate similarity metric like `LogLikelihoodSimilarity`. Precision and recall increase to over 22 percent in this case. Similar results are seen with `TanimotoCoefficientSimilarity`. These results seem better on the surface; what they say is that this sort of recommender engine is better at recommending those profiles that the user might already have encountered. If there were reason to believe users had, in fact, reviewed a large proportion of all profiles, their actual top ratings would be a strong indicator of what the right answers are. This isn't likely to be the case on a dating site with hundreds of thousands of profiles.

In other contexts, a high precision and recall figure may be important. Here, it doesn't seem to be as important. For our purposes, we move forward with the previous user-based recommender, with Euclidean distance similarity and a nearest-2 neighborhood, instead of opting to switch to one of these other recommenders.

5.2.5 Evaluating Performance

It's important to look at the runtime performance of our chosen recommender. Because it'll be queried in real time, it would do little good to produce a recommender that needs minutes to compute a recommendation!

The `LoadEvaluator` class can be used, as before, to assess the time required per recommendation. We ran this recommender on the data set with these flags

```
-server -d64 -Xmx2048m -XX:+UseParallelGC -XX:+UseParallelOldGC
```

and we found an average recommendation time of 218 milliseconds on our test machine. The application consumes only about a gigabyte of heap at runtime. Whether or not these values are acceptable or not will depend on application requirements and available hardware. These figures seem reasonable for many applications.

So far we've just applied standard Mahout recommenders to the data set at hand. We've had to do no customization. But creating the best recommender system for a particular data set or site inevitably requires taking advantage of all the information you can. This, in turn, requires some degree of customization and specialization of standard implementations, like those in Mahout, to make use of the particular attributes of the problem at hand. In the next section, we look at extending the existing Mahout implementations in small ways to take advantage of particular properties of the dating data, in order to increase the quality of our recommendations.

5.3 Injecting domain-specific information

So far the recommender hasn't taken advantage of any domain-specific knowledge. It hasn't taken advantage of the fact that the ratings are, specifically, people rating other people. It has used the user-profile rating data as if it were anything at all—ratings for books or cars or fruit. But it's often possible to incorporate additional information in the data to improve recommendations.

In the following subsections, we look at ways to incorporate one important piece of information in this data set that has so far been unused: gender. We create a custom `ItemSimilarity` metric based on gender, and we also look at how to avoid recommending users of an inappropriate gender.

5.3.1 Employing a custom item similarity metric

Because the gender of many profiles is given, you could create a simple similarity metric for pairs of profiles based only on gender. Profiles are items, so this would be an `ItemSimilarity` in the framework.

For example, call two male or two female profiles very similar and assign them a similarity of 1.0. Say the similarity between a male and female profile is –1.0. Finally, assign a 0.0 to profile pairs where the gender of one or both is unknown.

This idea is simple, perhaps overly simplistic. It would be fast, but it would discard all rating-related information from the metric computation. For the sake of experimentation, let's try it out with an item-based recommender, as follows.

Listing 5.2 A gender-based item similarity metric

```
public class GenderItemSimilarity implements ItemSimilarity {

  private final FastIDSet men;
  private final FastIDSet women;

  public GenderItemSimilarity(FastIDSet men, FastIDSet women) {
    this.men = men;
    this.women = women;
  }

  public double itemSimilarity(long profileID1, long profileID2) {
    Boolean profile1IsMan = isMan(profileID1);
    if (profile1IsMan == null) {
      return 0.0;
```

```
    }
    Boolean profile2IsMan = isMan(profileID2);
    if (profile2IsMan == null) {
      return 0.0;
    }
    return profile1IsMan == profile2IsMan ? 1.0 : -1.0;
  }

  public double[] itemSimilarities(long itemID1, long[] itemID2s) {
    double[] result = new double[itemID2s.length];
    for (int i = 0; i < itemID2s.length; i++) {
      result[i] = itemSimilarity(itemID1, itemID2s[i]);
    }
    return result;
  }
...
  private Boolean isMan(long profileID) {
    if (men.contains(profileID)) {
      return Boolean.TRUE;
    }
    if (women.contains(profileID)) {
      return Boolean.FALSE;
    }
    return null;
  }

  public void refresh(Collection<Refreshable> alreadyRefreshed) {
    // do nothing
  }

}
```

This `ItemSimilarity` metric can be paired with a standard `GenericItemBased-Recommender`, as before, and we can evaluate the recommender's accuracy. The concept is interesting, but the result here isn't better than with other metrics: 2.35. If more information were available, such as the interests and hobbies expressed on each profile, it could form the basis of a more meaningful similarity metric that might yield better results.

This example, however, illustrates the main advantage of item-based recommenders: it provides a means to incorporate information about items themselves, which is commonly available in recommender problems. From the evaluation results, you also perhaps noticed how this kind of recommender is fast when based on such an easy-to-compute similarity metric; on our test machine, recommendations were produced in about 15 milliseconds on average.

5.3.2 Recommending based on content

If you blinked, you might have missed it—that was an example of content-based recommendation in the last section. It had a notion of item similarity that *wasn't* based on user preferences, but that was based on *attributes* of the item itself. As we've said before, Mahout doesn't provide content-based recommendation implementations,

but it does provide expansion points and APIs that let you write code to deploy it within the framework.

It's a powerful addition to pure collaborative filtering approaches, which are based only on user preferences. You can usefully inject your knowledge about items (here, people) to augment the user preference data you have, and hopefully produce better recommendations.

Unfortunately the previous item similarity metric is specific to the problem domain at hand. This metric doesn't help with recommendations in other domains: recommending food, or movies, or travel destinations. That's why it's not part of the framework. But it's a feasible and powerful approach any time you have domain-specific knowledge, beyond user preferences, about how items are related.

5.3.3 *Modifying recommendations with IDRescorer*

You may have observed an optional, final argument to the `Recommender.recommend()` method of type `IDRescorer`; instead of calling `recommend(long userID, int howMany)`, you can call `recommend(long userID, int howMany, IDRescorer rescorer)`. Implementations of this interface show up in several parts of the Mahout recommender-related APIs. Implementations can transform values used in the recommender engine to other values based on some logic, or they can exclude an entity from consideration in some process. For example, an `IDRescorer` can be used to arbitrarily modify a `Recommender`'s estimated preference value for an item. It can also remove an item from consideration entirely.

Suppose you were recommending books to a user on an e-commerce site. The user is currently browsing mystery novels, so when recommending books to that user at that moment, you might wish to boost estimated preference values for all mystery novels. You may also wish to ensure that no out-of-stock books are recommended. An `IDRescorer` can help you do this. The following listing shows an `IDRescorer` implementation that encapsulates this logic in terms of some made-up implementation classes, like Genre, from this fictitious bookseller.

Listing 5.3 Example `IDRescorer` that omits out-of-stock books and boosts a genre

```
public class GenreRescorer implements IDRescorer {

  private final Genre currentGenre;

  public GenreRescorer(Genre currentGenre) {
    this.currentGenre = currentGenre;
  }

  public double rescore(long itemID, double originalScore) {
    Book book = BookManager.lookupBook(itemID);
    if (book.getGenre().equals(currentGenre)) {
      return originalScore * 1.2;
    }
    return originalScore;
  }
```

Assume BookManager exists

Boost estimate by 20%

Don't change anything else

```
  public boolean isFiltered(long itemID) {
    Book book = BookManager.lookupBook(itemID);
    return book.isOutOfStock();                    <- Filter not-in-stock books
  }
}
```

The `rescore()` method boosts the estimated preference value for mystery novels. The `isFiltered()` method demonstrates the other use of `IDRescorer`: it ensures that no out-of-stock books are considered for recommendation.

This was merely an example and isn't relevant to our dating site. Let's apply this idea with the extra data available: gender.

5.3.4 Incorporating gender in an IDRescorer

An `IDRescorer` can filter out items, or user profiles, for users whose gender may not be of romantic interest. An implementation might work by first guessing the user's preferred gender, by examining the gender of profiles rated so far. Then it could filter out profiles of the opposite gender, as seen in the next listing.

Listing 5.4 Gender-based rescoring implementation

```
public class GenderRescorer implements IDRescorer {

  private final FastIDSet men;
  private final FastIDSet women;
  private final FastIDSet usersRateMoreMen;      <- Cache which users rate more males
  private final FastIDSet usersRateLessMen;
  private final boolean filterMen;

  public GenderRescorer(FastIDSet men,
                        FastIDSet women,
                        FastIDSet usersRateMoreMen,
                        FastIDSet usersRateLessMen,
                        long userID, DataModel model)
      throws TasteException {
    this.men = men;
    this.women = women;
    this.usersRateMoreMen = usersRateMoreMen;
    this.usersRateLessMen = usersRateLessMen;
    this.filterMen = ratesMoreMen(userID, model);
  }

  public static FastIDSet[] parseMenWomen(File genderFile)
      throws IOException {                        <- Called separately later
    FastIDSet men = new FastIDSet(50000);
    FastIDSet women = new FastIDSet(50000);
    for (String line : new FileLineIterable(genderFile)) {
      int comma = line.indexOf(',');
      char gender = line.charAt(comma + 1);
      if (gender == 'U') {
        continue;
      }
      long profileID = Long.parseLong(line.substring(0, comma));
      if (gender == 'M') {
```

```
        men.add(profileID);
      } else {
        women.add(profileID);
      }
    }
    men.rehash();
    women.rehash();
    return new FastIDSet[] { men, women };
  }

  private boolean ratesMoreMen(long userID, DataModel model)
      throws TasteException {
    if (usersRateMoreMen.contains(userID)) {
      return true;
    }
    if (usersRateLessMen.contains(userID)) {
      return false;
    }
    PreferenceArray prefs = model.getPreferencesFromUser(userID);
    int menCount = 0;
    int womenCount = 0;
    for (int i = 0; i < prefs.length(); i++) {
      long profileID = prefs.get(i).getItemID();
      if (men.contains(profileID)) {
        menCount++;
      } else if (women.contains(profileID)) {
        womenCount++;
      }
    }
    boolean ratesMoreMen = menCount > womenCount;
    if (ratesMoreMen) {
      usersRateMoreMen.add(userID);
    } else {
      usersRateLessMen.add(userID);
    }
    return ratesMoreMen;
  }

  public double rescore(long profileID, double originalScore) {
    return isFiltered(profileID)
      ? Double.NaN : originalScore;
  }

  public boolean isFiltered(long profileID) {
    return filterMen ? men.contains(profileID) : women.contains(profileID);
  }

}
```

Re-optimize for fast access

Users rating men probably like male profiles

NaN for profiles to be excluded

A few things are happening in this code example. The `parseMenWomen()` method will parse gender.dat and create two sets of profile IDs—those that are known to be men and those known to be women. This is parsed separately from any particular instance of `GenderRescorer`, because these sets will be reused many times. The `ratesMoreMen()` method will be used to determine and remember whether a user seems to rate more male or female profiles. These results are cached in two additional sets.

Instances of this GenderRescorer will then simply filter out men, or women, as appropriate, by returning NaN from rescore(), or true from isFiltered().

This ought to have some small but helpful effect on the quality of the recommendations. Presumably, women who rate male profiles are *already* being recommended male profiles, because they'll be most similar to other women who rate male profiles. This mechanism will ensure this by filtering female profiles from the results. It'll cause the recommender to not even attempt to estimate these women's preference for *female* profiles because such an estimate is quite a guess, and likely wrong. Of course, the effect of this IDRescorer is limited by the quality of data available: the gender of only about half of the profiles is known.

5.3.5 Packaging a custom recommender

It will be useful for our purposes to wrap up the current recommender engine, plus the new IDRescorer, into one implementation. This will allow us, in section 5.5, to deploy one self-contained recommender engine.

The following listing shows a recommender implementation that contains the previous user-based recommender engine.

Listing 5.5 Complete recommender implementation for Líbímseti

```
public class LibimsetiRecommender implements Recommender {

  private final Recommender delegate;
  private final DataModel model;
  private final FastIDSet men;
  private final FastIDSet women;
  private final FastIDSet usersRateMoreMen;
  private final FastIDSet usersRateLessMen;

  public LibimsetiRecommender() throws TasteException, IOException {
    this(new FileDataModel(
      readResourceToTempFile("ratings.dat")));       <- Need readResourceToTempFile()
  }                                                      in production

  public LibimsetiRecommender(DataModel model)
      throws TasteException, IOException {
    UserSimilarity similarity =                          Construct user-based
        new EuclideanDistanceSimilarity(model);      <-  recommender
    UserNeighborhood neighborhood =
        new NearestNUserNeighborhood(2, similarity, model);
    delegate =
        new GenericUserBasedRecommender(model, neighborhood, similarity);
    this.model = model;
    FastIDSet[] menWomen = GenderRescorer.parseMenWomen(
        readResourceToTempFile("gender.dat"));
    men = menWomen[0];
    women = menWomen[1];
    usersRateMoreMen = new FastIDSet(50000);
    usersRateLessMen = new FastIDSet(50000);
  }
```

```
  public List<RecommendedItem> recommend(long userID, int howMany)
      throws TasteException {
    IDRescorer rescorer = new GenderRescorer(
        men, women, userID, usersRateMoreMen, usersRateLessMen,
        userID, model);
    return delegate.recommend(userID, howMany, rescorer);
  }

  public List<RecommendedItem> recommend(long userID,
                                         int howMany,
                                         IDRescorer rescorer)
      throws TasteException {
    return delegate.recommend(userID, howMany, rescorer);
  }

  public float estimatePreference(long userID, long itemID)
      throws TasteException {
    IDRescorer rescorer = new GenderRescorer(
        men, women, userID, usersRateMoreMen, usersRateLessMen,
        userID, model);
    return (float) rescorer.rescore(
        itemID, delegate.estimatePreference(userID, itemID));
  }

  public void setPreference(long userID, long itemID, float value)
      throws TasteException {
    delegate.setPreference(userID, itemID, value);
  }

  public void removePreference(long userID, long itemID)
      throws TasteException {
    delegate.removePreference(userID, itemID);
  }

  public DataModel getDataModel() {
    return delegate.getDataModel();
  }

  public void refresh(Collection<Refreshable> alreadyRefreshed) {
    delegate.refresh(alreadyRefreshed);
  }

}
```

Use GenderRescorer for all recommendations

Rescore estimated preferences

Delegate to underlying recommender

This is a tidy, self-contained packaging of the recommender engine. If you evaluate it, the result is about 1.18; this is virtually unchanged, and it's good to have this mechanism in place to avoid some seriously undesirable recommendations. Running time has increased to 500 milliseconds or so—the rescoring has added significant overhead. For our purposes here, this tradeoff is acceptable, and `LibimsetiRecommender` is the final implementation for this dating site.

5.4 Recommending to anonymous users

Another common issue you'll soon run into when creating a real production-ready recommender is what to do with users that aren't registered users yet. What can be done, for instance, for the new user browsing products in an e-commerce website?

This anonymous user has no browsing or purchase history, let alone an ID, as far as the site is concerned. The problem of making recommendations to such users, starting from no data, is called the *cold start problem*. It's nevertheless valuable to be able to recommend products to such a user.

One extreme approach is to not bother personalizing the recommendations. That is, when presented with a new user, present a general predefined list of products to recommend. It's simple, and usually better than nothing, but not an option we explore. Mahout is in the business of personalizing recommendations.

At the other extreme end of the spectrum, a site could promote such anonymous users to real users on their first visit, and assign an ID and track their activity based on a web session. This also works, though it potentially explodes the number of users, many of whom, by definition, may never return and for whom little information exists. This is also not an option we explore.

Instead, we examine two compromise approaches between these extremes: creating temporary users and treating all anonymous users as if they were one user.

5.4.1 Temporary users with PlusAnonymousUserDataModel

The recommender framework offers a simple way to temporarily add an anonymous user's information into the `DataModel`: by using the `PlusAnonymousUserDataModel` class. This approach treats anonymous users like real users, but only for as long as it takes to make recommendations. They're never added to or known to the real underlying `DataModel`. `PlusAnonymousUserDataModel` is a wrapper around any existing `DataModel` and is simply a drop-in replacement

The `PlusAnonymousUserDataModel` class has a spot for one temporary user, and it can hold preferences for one such user at a time. As such, a `Recommender` based on this class must only operate on one anonymous user at a time.

The next listing presents `LibimsetiWithAnonymousRecommender`, which extends the previous `LibimsetiRecommender` with a method that can recommend to an anonymous user. It takes preferences as input rather than a user ID, of course.

Listing 5.6 Anonymous user recommendation for Líbímseti

```
public class LibimsetiWithAnonymousRecommender
    extends LibimsetiRecommender {

  private final PlusAnonymousUserDataModel plusAnonymousModel;

  public LibimsetiWithAnonymousRecommender()
      throws TasteException, IOException {
    this(new FileDataModel(
        readResourceToTempFile("ratings.dat")));
  }

  public LibimsetiWithAnonymousRecommender(DataModel model)
      throws TasteException, IOException {
    super(new PlusAnonymousUserDataModel(model));       <- Wraps underlying DataModel
    plusAnonymousModel =
```

```
        (PlusAnonymousUserDataModel) getDataModel();
  }

  public synchronized List<RecommendedItem> recommend(
      PreferenceArray anonymousUserPrefs, int howMany)
      throws TasteException {
    plusAnonymousModel.setTempPrefs(anonymousUserPrefs);
    List<RecommendedItem> recommendations =
        recommend(PlusAnonymousUserDataModel.TEMP_USER_ID,
                  howMany, null);
    plusAnonymousModel.clearTempPrefs();
    return recommendations;
  }

  public static void main(String[] args) throws Exception {
    PreferenceArray anonymousPrefs =
        new GenericUserPreferenceArray(3);
    anonymousPrefs.setUserID(0,
        PlusAnonymousUserDataModel.TEMP_USER_ID);
    anonymousPrefs.setItemID(0, 123L);
    anonymousPrefs.setValue(0, 1.0f);
    anonymousPrefs.setItemID(1, 123L);
    anonymousPrefs.setValue(1, 3.0f);
    anonymousPrefs.setItemID(2, 123L);
    anonymousPrefs.setValue(2, 2.0f);
    LibimsetiWithAnonymousRecommender recommender =
        new LibimsetiWithAnonymousRecommender();
    List<RecommendedItem> recommendations =
        recommender.recommend(anonymousPrefs, 10);
    System.out.println(recommendations);
  }
// implementation of  readResourceToTempFile is omitted
// ...
}
```

Uses synchronization

Sets anonymous user's "ID"

Stores anonymous user prefs

This implementation otherwise works like any other recommender and can be used to recommend to real users as well.

5.4.2 *Aggregating anonymous users*

It's also possible to treat all anonymous users as if they were one user. This simplifies things. Rather than tracking potential users separately and storing their browsing histories individually, you could think of all such users as one big tire-kicking user. This depends upon the assumption that all such users behave similarly.

At any time, this technique can produce recommendations for the anonymous user, and it's fast. In fact, because the result is the same for all anonymous users, the set of recommendations can be stored and recomputed periodically instead of upon every request. In a sense, this variation nearly reduces to not personalizing recommendations at all, and to presenting anonymous users with a fixed set of recommendations.

5.5 Creating a web-enabled recommender

Creating a recommender that runs in your IDE is fine, but chances are you're interested in deploying this recommender in a real production application.

You may wish to deploy a recommender that you've designed and tested in Java and Mahout as a standalone component of your application architecture, rather than embed it inside your application's Java code. It's common for services to be exposed over the web, via simple HTTP or web service protocols like SOAP. In this scenario, a recommender is deployed as a web-accessible service, or as an independent component in a web container, or even as its own server process. This adds complexity, but it allows other applications written in other languages, or running on other machines, to access the service.

Fortunately, Mahout makes it simple to bundle your recommender implementation into a deployable WAR (web archive) file. Such a component can be readily deployed into any Java servlet container, such as Tomcat (http://tomcat.apache.org/) or Resin (http://www.caucho.com/resin/). The WAR file, illustrated in figure 5.3, wraps up your `Recommender` implementation and exposes it via a simple servlet-based HTTP service, `RecommenderServlet`, and as an Apache Axis-powered web service using SOAP over HTTP, `RecommenderService`.

5.5.1 Packaging a WAR file

The compiled code, plus data file, will need to be packaged into a JAR file before they can be deployed. Chances are you've already compiled this code with your IDE. Change to the book's examples directory and copy the data set's ratings.dat and gender.dat files into the src/main/resources directory, and then make a JAR file with this command:

```
mvn package
```

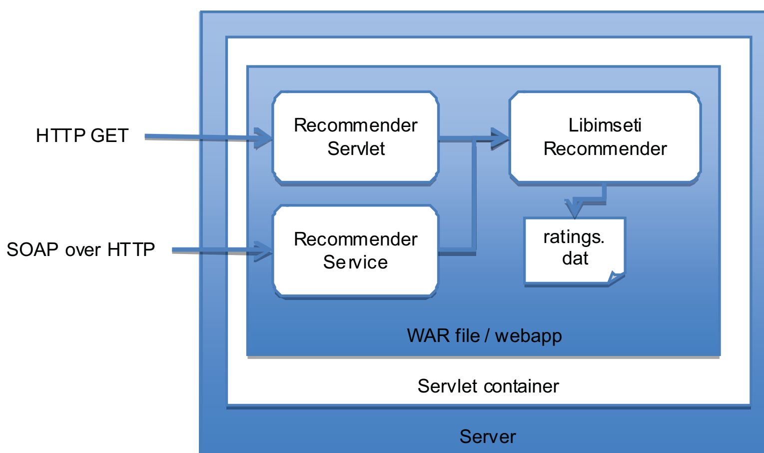


Figure 5.3 Automated WAR packaging of a recommender and deployment in a servlet container

That will put the result into the target/mia-0.1.jar file.

Then change to the Mahout distribution and the taste-web/ module directory, and copy target/mia-0.1.jar from the book's examples into the lib/ subdirectory. Also, edit recommender.properties to name your recommender as the one that will be deployed. If you're using the same Java package name as the sample code does, the right value is `mia.recommender.ch05.LibimsetiRecommender`.

Now execute `mvn package` again. You should find a .war file in the target/ subdirectory named mahout-taste-webapp-0.5.war (the version number may be higher if, by the time you read this, Mahout has published further releases). This file is suitable for immediate deployment in a servlet container like Tomcat. In fact, it can be dropped into Tomcat's webapps/ directory without further modification to produce a working web-based instance of your recommender.

> **NOTE** The name of the .war file will become part of the URL used to access the services; you may wish to rename it to something shorter like mahout.war.

5.5.2 Testing deployment

Alternatively, you can easily test this same web application containing your recommender without bothering to set up Tomcat by using Maven's built-in Jetty plugin. Jetty (http://www.mortbay.org/jetty/) is an embeddable servlet container, which serves a function similar to that of Tomcat or Resin.

Before firing up a test deployment, you'll need to ensure that your local Mahout installation has been compiled and made available to Maven. Execute `mvn install` from the top-level Mahout directory and take a coffee break, because this will cause Maven to download other dependencies, compile, and run tests, all of which takes 10 minutes or so. This only needs to be done once.

Having packaged the WAR file as discussed in the previous section, execute `export MAVEN_OPTS=-Xmx2048m` to ensure that Maven and Jetty have plenty of heap space available. Then, from the taste-web/ directory, execute `mvn jetty:run-war`. This will start up the web-enabled recommender services on port 8080 on your local machine. Startup could take some time, because Mahout will load and analyze data files.

In your web browser, navigate to the URL:

http://localhost:8080/RecommenderServlet?userID=3

to retrieve recommendations for user ID 3. This is precisely how an external application could access recommendations from your recommender engine, by issuing an HTTP `GET` request for this URL and parsing the simple text result: recommendations, one estimated preference value and item ID per line, with the best preference first (as shown in the next listing).

Listing 5.7 Output of a `GET` to `RecommenderServlet`

```
10.0    205930
10.0    156148
8.0     162783
```

```
7.5     208304
7.5     143604
7.0     210831
7.0     173483
4.5     163100
```

To explore the more formal SOAP-based web service API that's available, access http://localhost:8080/RecommenderService.jws?wsdl to see the WSDL (Web Services Definition Language) file that defines the input and output of this web service. It exposes a simplified version of the `Recommender` API. This web services description file can be consumed by most web service clients to automatically understand and provide access to the API.

If you're interested in trying the service directly in a browser, access http://localhost:8080/RecommenderService.jws?method=recommend&useID=1&howMany=10 to see the SOAP-based reply from the service. It's the same set of results, just presented as a SOAP response (see figure 5.4).

```
- <soapenv:Envelope>
  - <soapenv:Body>
    - <recommendResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      - <recommendReturn soapenc:arrayType="xsd:string[][10]" xsi:type="soapenc:Array">
        - <recommendReturn soapenc:arrayType="xsd:string[2]" xsi:type="soapenc:Array">
            <recommendReturn xsi:type="xsd:string">10.0</recommendReturn>
            <recommendReturn xsi:type="xsd:string">220429</recommendReturn>
          </recommendReturn>
        - <recommendReturn soapenc:arrayType="xsd:string[2]" xsi:type="soapenc:Array">
            <recommendReturn xsi:type="xsd:string">10.0</recommendReturn>
            <recommendReturn xsi:type="xsd:string">174211</recommendReturn>
          </recommendReturn>
```

Figure 5.4 Browser rendering of the SOAP response from `RecommenderService`

Normally, at this point, you would be sanity-checking the results. Put yourself in your users' shoes—do the recommendations make sense? Here, it's not possible to know who the users are or what the profiles are like, so there's no intuitive interpretation of the results. This wouldn't be true when developing your own recommender engine, where a look at the actual recommendations would likely give insight into problems or opportunities for refinement. This would lead to more cycles of experimentation and modification to make the results match the most appropriate answers for your problem domain.

5.6 Updating and monitoring the recommender

Now you have a live web-based recommender service running, but it's not a static, fixed system that's run and then forgotten. It's a dynamic server, ingesting new information and returning answers in real time, and like any production system, it's natural to think about how the service will be updated and monitored.

Of course, the data on which recommendations are based changes constantly in a real recommender engine system. Standard `DataModel` implementations will automatically use the most recent data available from your underlying data source, so, at a

high level, there's nothing special that needs to be done to cause the recommender engine to incorporate new data. For example, if you had based your recommender engine on data in a database, by using a `JDBCDataModel`, then just by updating the underlying database table with new data, the recommender engine would begin using that data.

But for performance, many components cache information and intermediate computations. These caches update eventually, but this means that new data doesn't immediately affect recommendations. It's possible to force all caches to clear by calling `Recommender.refresh()`, and this can be done by invoking the `refresh` method on the SOAP-based interface that's exposed by the web application harness. If needed, this can be invoked by other parts of your enterprise architecture.

File-based preference data, accessed via a `FileDataModel`, deserves some special mention. The file can be updated or overwritten in order to deploy updated information; `FileDataModel` will shortly thereafter notice the update and reload the file.

Reloading the data file can be slow and memory-intensive, as both the old and new model will be in memory at the same time. Now is a good time to recall *update files*, introduced in chapter 3. Instead of replacing or updating the main data file, it's more efficient to add update files representing recent updates. The update files are like diffs, and when placed in the same directory as the main data file and named appropriately, they'll be detected and applied quickly to the in-memory representation of the preference data.

For example, an application might, each hour, locate all preference data created, deleted, or changed in the last 60 or more minutes, create an update file, and copy it alongside the main data file. Recall also that, for efficiency, all of these files can be compressed.

Monitoring the health of this recommender service is straightforward, even if support for monitoring is outside the scope of Mahout itself. Any monitoring tool that can check the health of a web-based service, accessed via HTTP, can easily check that the recommender service is live by accessing the service URL and verifying that a valid answer is returned. Such tools can and should also monitor the time it takes to answer these requests and create an alert if performance suddenly degrades. Normally, the time it takes to compute a recommendation is quite consistent and shouldn't vary greatly.

5.7 Summary

In this chapter, we took an in-depth look at a real, large data set made available from the Czech dating site Líbímseti. It provides 17 million ratings of over a hundred thousand profiles from over a hundred thousand users. We set out to create a recommender for this site that could recommend profiles, or people, to its users.

We tried most of the recommender approaches seen so far with this data set and used evaluation techniques to choose an implementation that seemed to produce the

best recommendations: a user-based recommender using a Euclidean distance-based similarity metric and a nearest-2 neighborhood definition.

From there, we explored mixing in additional information from the data set: the gender of the users, which is featured in many of the profiles. We tried creating an item similarity metric based on this data. We met the `IDRescorer` interface, a practical tool that can be used to modify results in ways specific to a problem domain. We achieved a small improvement by using an `IDRescorer` to take account of gender and exclude recommendations from the gender that doesn't apparently interest the user.

Having tested performance and found it acceptable (about 500 ms per recommendation), we constructed a deployable version of the recommender engine and automatically created a web-enabled application around it using Mahout. We briefly examined how to deploy and access this component via HTTP and SOAP.

Finally, we reviewed how to update, at runtime, the recommender's underlying data.

This concludes our journey from data to production-ready recommender service. This implementation can comfortably digest the data set of 17 million ratings on one machine and produce recommendations in real time. But what happens when the data outgrows one machine? In the next chapter, we examine how to handle a much larger data set with Hadoop.

6 Distributing recommendation computations

This chapter covers

- Analyzing a massive data set from Wikipedia
- Producing recommendations with Hadoop and distributed algorithms
- Pseudo-distributing existing nondistributed recommenders

This book has looked at increasingly large data sets: from 10s of preferences, to 100,000, to 10 million, and then 17 million. But this is still only medium-sized in the world of recommenders. This chapter ups the ante again by tackling a larger data set of 130 million preferences in the form of article-to-article links from Wikipedia's massive corpus.[1] In this data set, the articles are both the users and the

[1] Readers of earlier drafts will recall that the subject of this chapter was the Netflix Prize data set. That data set is no longer officially distributed, for legal reasons, and so is no longer a suitable example data set.

items, which also demonstrates how recommenders can be usefully applied, with Mahout, to less conventional contexts.

Although 130 million preferences is still a manageable size for demonstration purposes, it's of such a scale that a single machine would have trouble processing recommendations from it in the way we've presented so far. It calls for a new species of recommender algorithm, using a distributed computing approach from Mahout based on the MapReduce paradigm and Apache Hadoop.

We start by examining the Wikipedia data set and understanding what it means to distribute a recommender computation. You'll learn how a simple distributed recommender system is designed in a distributed environment, because it differs greatly from nondistributed implementations. You'll also see how, in Mahout, this design translates into an implementation based on MapReduce and Hadoop. Finally, you'll have a first occasion to run a complete Hadoop-based recommender job and see the results.

6.1 Analyzing the Wikipedia data set

We'll begin here by taking a look at the Wikipedia data set, but you'll quickly see why it's difficult to proceed as before, due to issues of scale. We'll shortly have to back up and take a look at distributing the computation to proceed any further.

Wikipedia (http://wikipedia.org) is a well-known online encyclopedia, whose contents can be edited and maintained by users. It reports that in May 2010 it contained over 3.2 million articles written in English alone. The Freebase Wikipedia Extraction project (http://download.freebase.com/wex/) estimates the size of just the English articles to be about 42 GB. Being web-based, Wikipedia articles can and do link to one another. It's these links that are of interest. Think of articles as users, and the articles that an article points to as items that the source article likes.

Fortunately, it's not necessary to download Freebase's well-organized Wikipedia extract and parse out all these links. Researcher Henry Haselgrove has already extracted the article links and published just this information at http://users.on.net/~henry/home/wikipedia.htm. This further filters out links to ancillary resources like article discussion pages, images, and such. This data set has also represented articles by their numeric IDs rather than titles, which is helpful, because Mahout treats all users and items as numeric IDs.

Before continuing, download and extract the links-simple-sorted.zip file from Haselgrove's page. This data set contains 130,160,392 links from 5,706,070 articles to 3,773,865 distinct other articles. Note that there are no explicit preferences or ratings; there are just associations from articles to articles. These are Boolean preferences in the language of the framework. Associations are one way; a link from A to B doesn't imply any association from B to A. There aren't significantly more items than users or vice versa, so neither a user-based nor item-based algorithms suggests itself as better from a performance perspective. If we use an algorithm that involves a similarity metric, one that doesn't depend on preference values is appropriate, like `LogLikelihoodSimilarity`.

What is the meaning of the data, intuitively, and what is it reasonable to expect from the recommendations? A link from article A to B implies that B provides information related to A, typically background information on entities or ideas referenced in the article. A recommender system built on this data will recommend articles that are pointed to by other articles that also point to some of the same articles that A points to. These other articles might be interpreted as articles that A should link to but doesn't. They could be articles that are simply also of interest to a reader of A. In some cases, the recommendations may reveal interesting or serendipitous associations that aren't even implied by article A.

6.1.1 Struggling with scale

Deploying a nondistributed recommender engine based on this data could prove difficult. The data alone would consume about 2 GB of JVM heap space with Mahout, and overall heap space would likely need to be 2.5 GB. On some 32-bit platforms and JVMs, this actually exceeds the maximum heap size that can be selected, which means a 64-bit machine would be required, if not immediately then soon. Depending on the algorithm, recommendation time could increase to over one second, which begins to be a long time for a real-time recommender engine supporting a modern web application.

With enough hardware, this could perform acceptably. But what happens when the input grows to a few billion preferences, and heap requirements top 32 GB? And beyond that? For a time, one could combat scale by throwing out progressively more of the "noise" data to keep its size down. But judging what is noise begins to be a problem of accuracy and scale in its own right.

It's unfashionable these days to be unable to cope with data beyond some scale, to have some hard limit on what a system can handle. Computing resources are readily available in large quantities; the problem here is putting enough computing resources into one box. It's disproportionately expensive to make a large machine even larger, as compared to obtaining more small machines. This massive single machine becomes a single point of failure. And it may be hard to find any efficient way to take advantage of the single machine's expensive power when it's not in use by the recommender engine process.

The size of this Wikipedia link data set represents about the practical upper limit of how much data can be thrown at a Mahout-based, nondistributed, real-time recommender on reasonable server hardware—and it's not even that big by modern standards. Beyond this scale, a new approach is needed.

6.1.2 Evaluating benefits and drawbacks of distributing computations

A solution lies in using many small machines, not one big one, for all the reasons that using one big machine is undesirable. An organization may already own and operate many small machines that aren't fully utilized, and whose extra capacity could be used toward computing recommendations, as illustrated in figure 6.1. Furthermore, the resources of many machines are readily available these days through cloud computing providers like Amazon's EC2 service (http://aws.amazon.com).

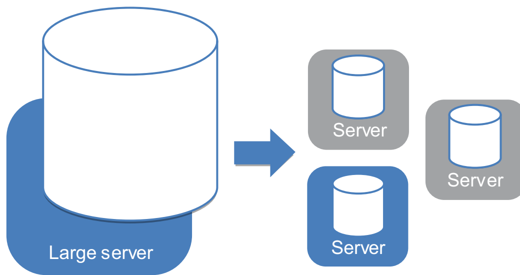


Figure 6.1 Distributed computation helps by breaking up a problem that's too big for one server into pieces that several smaller servers can handle.

Distributing a recommendation computation radically changes the recommender engine problem. Every algorithm so far has computed recommendations as a function of, in theory, every single preference value. To recommend new links for a single article from the Wikipedia link data set, the algorithm would need access to all article-to-article links, because the computation could draw on any of them. But at large scale, access to all or even most of the data isn't possible at any one time, because of its sheer size. All of the approaches we've looked at so far go out the window, at least in anything like their current form. Distributed recommender engine computations are a whole new ball game.

To be clear, distributing a computation doesn't make it more efficient. On the contrary, it usually makes it require significantly more resources. For instance, moving data between many small machines consumes network resources. The computation must often be structured in a way that involves computing and storing many intermediate results, which could take significant processing time to serialize, store, and deserialize later. The software that orchestrates these operations consumes nontrivial memory and processing power.

It should be noted that such large, distributed computations are necessarily performed offline, not in real time in response to user requests. Even small computations of this form take at least minutes, not milliseconds, to complete. Commonly, recommendations would be recomputed at regular intervals, stored, and returned to the user at runtime.

But these approaches offer a means to complete a recommender engine computation at scales where nondistributed computations can't even start due to the lack of resources on a single machine. Because distributed computations can leverage bits of resources from many machines, they offer the possibility of using spare, unused resources from existing machines rather than dedicated machines. Finally, distributed computations can allow a computation to complete earlier—even though it might take more raw processing time. Say a distributed computation takes twice as much CPU time as its nondistributed counterpart. If 10 CPUs work on the computation, it'll complete five times faster than a nondistributed version, which can only take advantage of one machine's resources.

6.2 Designing a distributed item-based algorithm

For problems of this scale, it's desirable and necessary to deploy a distributed approach to produce recommendations. First, this section will sketch out a distributed variation on the item-based recommender approach we looked at earlier. It will be similar in some ways to that nondistributed item-based recommender. But it'll certainly look different, because the nondistributed algorithm doesn't fully translate to the distributed world.

6.2.1 Constructing a co-occurrence matrix

The algorithm is best explained, and implemented, in terms of simple matrix operations. If the last time you touched matrices was in a math textbook years ago, don't worry: the trickiest operation you'll need to recall is matrix multiplication. There will be no determinants, row reduction, or eigenvalues here.

Recall that the item-based implementations so far rely on an `ItemSimilarity` implementation, which provides some notion of the degree of similarity between any pair of items. Imagine computing a similarity for every pair of items and putting the results into a giant matrix. It would be a square matrix, with the number of rows and columns equal to the number of items in the data model. Each row (and each column) would express similarities between one particular item and all other items. It will be useful to think of these rows and columns as vectors, in fact. It would be symmetric across the diagonal as well; because the similarity between items X and Y is the same as the similarity between items Y and X, the entry in row X and column Y would equal the entry in row Y and column X.

> **NOTE** This matrix describes associations between items, and has nothing to do with users. It's not a user-item matrix. That sort of matrix would not be symmetric; its number of rows and columns would match the count of users and items, which are not the same.

Something like this is needed for the algorithm: a *co-occurrence matrix*. Instead of computing the *similarity* between every pair of items, it'll compute the number of times each pair of items occurs together in some user's list of preferences, in order to fill out the matrix. For instance, if there are 9 users who express some preference for both items X and Y, then X and Y co-occur 9 times. Two items that never appear together in any user's preferences have a co-occurrence of 0. And, conceptually, each item co-occurs with itself every time any user expresses a preference for it, though this count won't be useful.

Co-occurrence is like similarity; the more two items turn up together, the more related or similar they probably are. The co-occurrence matrix plays a role like that of `ItemSimilarity` in the nondistributed item-based algorithm.

Producing the matrix is a simple matter of counting. Note that the entries in the matrix aren't affected by preference values. These values will enter the computation

Table 6.1 The co-occurrence matrix for items in a simple example data set. The first row and column are labels and not part of the matrix.

	101	102	103	104	105	106	107
101	5	3	4	4	2	2	1
102	3	3	3	2	1	1	0
103	4	3	4	3	1	2	0
104	4	2	3	4	2	2	1
105	2	1	1	2	2	1	1
106	2	1	2	2	1	2	0
107	1	0	0	1	1	0	1

later. Table 6.1 shows the co-occurrence matrix for the small example set of preference values that have been used throughout the book.

As advertised, the matrix is symmetric across the diagonal. There are 7 items, and the matrix is a 7 x 7 square matrix. The values on the diagonal, it turns out, won't be of use to the algorithm, but they're included for completeness.

6.2.2 Computing user vectors

The next step in converting the previous recommender approaches to a matrix-based distributed computation is to conceive of a user's preferences as a vector. You already did this, in a way, when discussing the Euclidean distance–based similarity metric, where users were thought of as points in space, and similarity was based on the distance between them.

Likewise, in a data model with n items, user preferences are like a vector over n dimensions, with one dimension for each item. The user's preference values for items are the values in the vector. Items that the user expresses no preference for map to a 0 value in the vector. Such a vector is typically quite sparse, and mostly zeroes, because users typically express a preference for only a small subset of all items.

For example, in the small example data set, user 3's preferences correspond to the vector [2.0, 0.0, 0.0, 4.0, 4.5, 0.0, 5.0]. To produce recommendations, each user needs such a vector.

6.2.3 Producing the recommendations

To compute recommendations for user 3, merely multiply this vector, as a column vector, with the co-occurrence matrix, as shown in table 6.2.

Take a moment to review how matrix multiplication works, if needed (http://en.wikipedia.org/wiki/Matrix_multiplication). The product of the co-occurrence matrix and a user vector is itself a vector whose dimension is equal to the number of items. The values in this resulting vector, R, lead us directly to recommendations: the highest values in R correspond to the best recommendations.

Table 6.2 Multiplying the co-occurrence matrix with user 3's preference vector (U3) to produce a vector that leads to recommendations, R

	101	102	103	104	105	106	107		U3		R
101	*5*	*3*	*4*	*4*	*2*	*2*	*1*		*2.0*		*40.0*
102	3	3	3	2	1	1	0		0.0		18.5
103	**4**	**3**	**4**	**3**	**1**	**2**	**0**	**X**	**0.0**	**=**	**24.5**
104	*4*	*2*	*3*	*4*	*2*	*2*	*1*		*4.0*		*40.0*
105	*2*	*1*	*1*	*2*	*2*	*1*	*1*		*4.5*		*26.0*
106	2	1	2	2	1	2	0		0.0		16.5
107	*1*	*0*	*0*	*1*	*1*	*0*	*1*		*5.0*		*15.5*

Table 6.2 shows this multiplication for user 3 in the small example data set, and the resulting vector R. It's possible to ignore the values in rows of R corresponding to items 101, 104, 105, and 107 (shown in italic) because these aren't eligible for recommendation: user 3 already expresses a preference for these items. Of the remaining items, the entry for item 103 is highest, with a value of 24.5, and would therefore be the top recommendation, followed by 102 and 106.

6.2.4 Understanding the results

Let's pause to understand what happened in the preceding section. Why do higher values in R correspond to better recommendations? Computing each entry in R is analogous to computing an estimated preference for one item, but why is that value like an estimated preference?

Recall that computing, for example, the third entry in R entails computing the dot product between the third row vector of the matrix, and column vector U3. This is the sum of the products of each corresponding pair of entries in the vectors:

```
4(2.0) + 3(0.0) + 4(0.0) + 3(4.0) + 1(4.5) + 2(0.0) + 0(5.0) = 24.5
```

That third row contains co-occurrences between item 103 and all other items. Intuitively, if item 103 co-occurs with many items that user 3 expresses a preference for, then it's probably something that user 3 would like. The preceding formula sums the products of co-occurrences and preference values. When item 103's co-occurrences overlap a lot with highly preferred items, the sum contains products of large co-occurrences and large preference values. That makes the sum larger, which is the value of the entry in R. This is why larger values in R correspond to good recommendations.

Note that the values in R don't represent an estimated preference value—they're far too large, for one. These could be normalized into estimated preference values with some additional computation, if desired. But for our purposes here, normalization doesn't matter, because the ordering of recommendations is the important thing, not the exact values on which the ordering depends.

6.2.5 Towards a distributed implementation

This is all very interesting, but what about this algorithm is more suitable for large-scale distributed implementations?

It's that the elements of this algorithm each involve only a subset of all data at any one time. For example, creating user vectors is merely a matter of collecting all preference values for one user and constructing a vector. Counting co-occurrences only requires examining one vector at a time. Computing the resulting recommendation vector only requires loading one row or column of the matrix at a time. Further, many elements of the computation just rely on collecting related data into one place efficiently—for example, creating user vectors from all the individual preference values.

The MapReduce paradigm was designed for computations with exactly these features.

6.3 Implementing a distributed algorithm with MapReduce

Now it's possible to translate the algorithm into a form that can be implemented with MapReduce and Apache Hadoop. Hadoop, as noted before, is a popular distributed computing framework that includes two components of interest: the Hadoop Distributed Filesystem (HDFS), and an implementation of the MapReduce paradigm.

The subsections that follow will introduce, one by one, the several MapReduce stages that come together into a pipeline that makes recommendations. Each individually does a little bit of the work. We'll look at the inputs, outputs, and purpose of each stage. Be prepared for plenty of reading; even this simple recommender algorithm will take five MapReduce stages, and this is a simplified form of how it exists in Mahout! By the end of this section you'll have seen a complete end-to-end recommender system based on Hadoop.


No. 6

Note that this chapter will use the Hadoop APIs found in version 0.20.2 of the framework. The code that follows can be found in its complete form within Mahout, and it should be runnable with Hadoop 0.20.2 or later versions of the 0.20.x branch. In particular, refer to `org.apache.mahout.cf.taste.hadoop.item.RecommenderJob`, which invokes the actual implementation of all of the following processes.

6.3.1 Introducing MapReduce

MapReduce is a way of thinking about and structuring computations in a way that makes them amenable to distributing over many machines. The shape of a MapReduce computation is as follows:

1. Input is assembled in the form of many key-value (K1,V1) pairs, typically as input files on an HDFS instance.
2. A map function is applied to each (K1,V1) pair, which results in zero or more key-value pairs of a different kind (K2,V2).
3. All V2 for each K2 are combined.
4. A reduce function is called for each K2 and all its associated V2, which results in zero or more key-value pairs of yet a different kind (K3,V3), output back to HDFS.

This may sound like an odd pattern for a computation, but many problems can be fit into this structure, or a series of them chained together. Problems framed in this way can then be efficiently distributed with Hadoop and HDFS.

Those unfamiliar with Hadoop should probably read and perhaps run Hadoop's short tutorial, found at http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html in the current 0.20.2 release, which illustrates the basic operation of a MapReduce job in Hadoop.

6.3.2 Translating to MapReduce: generating user vectors

In this case, the computation begins with the links data file as input. Its lines aren't of the form `userID,itemID,preference`. Instead they're of the form `userID: itemID1 itemID2 itemID3 ...`. This file is placed onto an HDFS instance in order to be available to Hadoop—more on how this is done a few sections later.

The first MapReduce will construct user vectors:

- Input files are treated as (`Long`,`String`) pairs by the framework, where the `Long` key is a position in the file and the `String` value is the line of the text file. For example, `239 / 98955: 590 22 9059`
- Each line is parsed into a user ID and several item IDs by a map function. The function emits new key-value pairs: a user ID mapped to item ID, for each item ID. For example, `98955 / 590`
- The framework collects all item IDs that were mapped to each user ID together.
- A reduce function constructors a `Vector` from all item IDs for the user, and outputs the user ID mapped to the user's preference vector. All values in this vector are 0 or 1. For example, `98955 / [590:1.0, 22:1.0, 9059:1.0]`

An implementation of this idea can be found in listings 6.1 and 6.2, which implement both Hadoop's MapReduce `Mapper` and `Reducer` interfaces. It's typical of MapReduce computations to have an implementation consist of a related pair of classes like this. These are all that are needed to implement the previously described process; Hadoop will take care of the rest.

Listing 6.1 A mapper that parses Wikipedia link files

```
public class WikipediaToItemPrefsMapper
    extends Mapper<LongWritable,Text,VarLongWritable,VarLongWritable> {

  private static final Pattern NUMBERS = Pattern.compile("(\\d+)");

  public void map(LongWritable key,
                  Text value,
                  Context context)
      throws IOException, InterruptedException {
    String line = value.toString();
    Matcher m = NUMBERS.matcher(line);
    m.find();                                            Locate user ID
    VarLongWritable userID =
        new VarLongWritable(Long.parseLong(m.group()));
    VarLongWritable itemID = new VarLongWritable();
```

```
    while (m.find()) {
      itemID.set(Long.parseLong(m.group()));
      context.write(userID, itemID);
    }
  }
}
```

Emit user-item pair for each item ID

Listing 6.2 Reducer which produces Vectors from a user's item preferences

```
public class WikipediaToUserVectorReducer extends
  Reducer<VarLongWritable,VarLongWritable,VarLongWritable,VectorWritable> {
  public void reduce(VarLongWritable userID,
                     Iterable<VarLongWritable> itemPrefs,
                     Context context)
      throws IOException, InterruptedException {
    Vector userVector = new RandomAccessSparseVector(
        Integer.MAX_VALUE, 100);
    for (VarLongWritable itemPref : itemPrefs) {
      userVector.set((int)itemPref.get(), 1.0f);
    }
    context.write(userID, new VectorWritable(userVector));
  }
}
```

Iterate over item-preference pairs for user

Set dimension "item ID" to item's preference value

These are simplified versions of the real implementations in Mahout, for illustration. They don't include optimizations and configuration options, but they would run and produce usable output.

6.3.3 *Translating to MapReduce: calculating co-occurrence*

The next phase of the computation is another MapReduce that uses the output of the first MapReduce to compute co-occurrences.

1. Input is user IDs mapped to `Vector`s of user preferences—the output of the last MapReduce. For example, `98955 / [590:1.0,22:1.0,9059:1.0]`
2. The map function determines all co-occurrences from one user's preferences, and emits one pair of item IDs for each co-occurrence—item ID mapped to item ID. Both mappings, from one item ID to the other and vice versa, are recorded. For example, `590 / 22`
3. The framework collects, for each item, all co-occurrences mapped from that item.
4. The reducer counts, for each item ID, all co-occurrences that it receives and constructs a new `Vector` that represents all co-occurrences for one item with a count of the number of times they have co-occurred. These can be used as the rows—or columns—of the co-occurrence matrix. For example,
   `590 / [22:3.0,95:1.0,...,9059:1.0,...]`

The output of this phase is in fact the co-occurrence matrix. Again, listings 6.3 and 6.4 provide a simplified look at how this is implemented in Mahout on top of Hadoop. Again there is a pair of related implementations, of `Mapper` and `Reducer`.

Listing 6.3 Mapper component of co-occurrence computation

```
public class UserVectorToCooccurrenceMapper extends
    Mapper<VarLongWritable,VectorWritable,IntWritable,IntWritable> {
  public void map(VarLongWritable userID,
                  VectorWritable userVector,
                  Context context)
      throws IOException, InterruptedException {
    Iterator<Vector.Element> it =
      userVector.get().iterateNonZero();
    while (it.hasNext()) {
      int index1 = it.next().index();
      Iterator<Vector.Element> it2 =
        userVector.get().iterateNonZero();
      while (it2.hasNext()) {
        int index2 = it2.next().index();
        context.write(new IntWritable(index1),
                      new IntWritable(index2));
      }
    }  }
```

- Only iterate over nonzero elements
- Record count of I

Listing 6.4 Reducer component of co-occurrence computation

```
public class UserVectorToCooccurrenceReducer extends
    Reducer<IntWritable,IntWritable,IntWritable,VectorWritable> {
  public void reduce(IntWritable itemIndex1,
                  Iterable<IntWritable> itemIndex2s,
                  Context context)
      throws IOException, InterruptedException {
    Vector cooccurrenceRow =
      new RandomAccessSparseVector(Integer.MAX_VALUE, 100);
    for (IntWritable intWritable : itemIndex2s) {
      int itemIndex2 = intWritable.get();
      cooccurrenceRow.set(
        itemIndex2,
        cooccurrenceRow.get(itemIndex2) + 1.0);
    }
    context.write(
        itemIndex1,
        new VectorWritable(cooccurrenceRow));
  }

}
```

- Accumulate counts for items I, 2
- Record entire item I vector

6.3.4 Translating to MapReduce: rethinking matrix multiplication

It's now possible to use MapReduce to multiply the user vectors computed in step 1, and the co-occurrence matrix from step 2, to produce a recommendation vector from which the algorithm can derive recommendations.

But the multiplication will proceed in a different way that's more efficient here—a way that more naturally fits the shape of a MapReduce computation. The algorithm won't perform conventional matrix multiplication, wherein each row is multiplied against the user vector (as a column vector), to produce one element in the result R:

```
for each row i in the co-occurrence matrix
    compute dot product of row vector i with the user vector
    assign dot product to ith element of R
```

Why depart from the algorithm we all learned in school? The reason relates purely to performance, and this is a good opportunity to examine the kind of thinking necessary to achieve performance at scale when designing large matrix and vector operations. The conventional algorithm necessarily touches the entire co-occurrence matrix, because it needs to perform a vector dot product with each row. Anything that touches the entire input is "bad" here, because the input may be staggeringly large and not even available locally. Instead, note that matrix multiplication can be accomplished as a function of the co-occurrence matrix *columns*:

```
assign R to be the zero vector
for each column i in the co-occurrence matrix
    multiply column vector i by the ith element of the user vector
    add this vector to R
```

Take a moment to convince yourself that this is also a correct way to define this matrix multiplication, with a small example perhaps. So far, this isn't an improvement: it also touches the entire co-occurrence matrix, by column.

But note that wherever element i of the user vector is 0, the loop iteration can be skipped entirely, because the product will be the zero vector and doesn't affect the result. So this loop need only execute for each nonzero element of the user vector. The number of columns loaded will be equal to the number of preferences that the user expresses, which is far smaller than the total number of columns when the user vector is sparse.

Expressed this way, the algorithm can distribute the computation efficiently. Column vector i can be output along with all the elements it needs to be multiplied against. The products can be computed and saved independently of the handling of all other column vectors.

6.3.5 Translating to MapReduce: matrix multiplication by partial products

The columns of the co-occurrence matrix are available from an earlier step. Because the matrix is symmetric, the rows and columns are identical, so this output can be viewed as either rows or columns, conceptually. The columns are keyed by item ID, and the algorithm must multiply each by every nonzero preference value for that item, across all user vectors. That is, it must map item IDs to a user ID and preference value, and then collect them together in a reducer. After multiplying the co-occurrence column by each value, it produces a vector that forms part of the final recommender vector R for one user.

The difficult part here is that two different kinds of data are joined in one computation: co-occurrence column vectors and user preference values. This isn't by nature possible in Hadoop, because values in a reducer can be of one `Writable` type only. A

clever implementation can get around this by crafting a `Writable` that contains either one or the other type of data: a `VectorOrPrefWritable`. Although this can be viewed as a hack, it may be valuable or necessary in designing a distributed computation to bend some rules to achieve an elegant, efficient computation.

The mapper phase here will actually contain two mappers, each producing different types of reducer input:

- Input for mapper 1 is the co-occurrence matrix: item IDs as keys, mapped to columns as `Vector`s. For example, `590 / [22:3.0,95:1.0,...,9059:1.0,...]`

  The map function simply echoes its input, but with the `Vector` wrapped in a `VectorOrPrefWritable`.
- Input for mapper 2 is again the user vectors: user IDs as keys, mapped to preference `Vector`s. For example, `98955 / [590:1.0,22:1.0,9059:1.0]`

  For each nonzero value in the user vector, the map function outputs an item ID mapped to the user ID and preference value, wrapped in a `VectorOrPrefWritable`. For example, `590 / [98955:1.0]`

  The framework collects together, by item ID, the co-occurrence column and all user ID–preference value pairs.

  The reducer collects this information into one output record and stores it.

The following listing shows the co-occurrence columns being wrapped in a `VectorOrPrefWritable`.

Listing 6.5 Wrapping co-occurrence columns

```
public class CooccurrenceColumnWrapperMapper extends
    Mapper<IntWritable,VectorWritable,
           IntWritable,VectorOrPrefWritable> {
  public void map(IntWritable key,
                  VectorWritable value,
                  Context context) throws IOException, InterruptedException {
    context.write(key, new VectorOrPrefWritable(value.get()));
  }
}
```

In the next listing, user vectors are split into their individual preference values and output, mapped by item ID rather than user ID.

Listing 6.6 Splitting user vectors

```
public class UserVectorSplitterMapper extends
    Mapper<VarLongWritable,VectorWritable,
           IntWritable,VectorOrPrefWritable> {
  public void map(VarLongWritable key,
                  VectorWritable value,
                  Context context) throws IOException, InterruptedException {
    long userID = key.get();
    Vector userVector = value.get();
    Iterator<Vector.Element> it = userVector.iterateNonZero();
    IntWritable itemIndexWritable = new IntWritable();
```

```
    while (it.hasNext()) {
      Vector.Element e = it.next();
      int itemIndex = e.index();
      float preferenceValue = (float) e.get();
      itemIndexWritable.set(itemIndex);
      context.write(itemIndexWritable,
                    new VectorOrPrefWritable(userID, preferenceValue));
    }
  }
}
```

Technically speaking, there is no real `Reducer` following these two `Mappers`; it's no longer possible to feed the output of two `Mappers` into a `Reducer`. Instead they're run separately, and the output is passed through a no-op `Reducer` and saved in two locations. These two locations can be used as input to another MapReduce, whose `Mapper` does nothing as well, and whose `Reducer` collects together a co-occurrence column vector for an item and all users and preference values for that item into a single entity called `VectorAndPrefsWritable`. `ToVectorAndPrefReducer` implements this; for brevity, this detail is omitted.

With columns of the co-occurrence matrix and user preferences in hand, both keyed by item ID, the algorithm proceeds by feeding both into a mapper that will output the product of the column and the user's preference, for each given user ID. This step is shown in listing 6.7.

1. Input to the mapper is all co-occurrence matrix columns and user preferences by item. For example, `590 / [22:3.0,95:1.0,...,9059:1.0,...]` and `590 / [98955:1.0]`
2. The mapper outputs the co-occurrence column for each associated user times the preference value. For example, `590 / [22:3.0,95:1.0,...,9059:1.0,...]`
3. The framework collects these partial products together, by user.
4. The reducer unpacks this input and sums all the vectors, which gives the user's final recommendation vector (call it R). For example, `590 / [22:4.0,45:3.0,95:11.0,...,9059:1.0,...]`

Listing 6.7 Computing partial recommendation vectors

```
public class PartialMultiplyMapper extends
    Mapper<IntWritable,VectorAndPrefsWritable,
           VarLongWritable,VectorWritable> {
  public void map(IntWritable key,
                  VectorAndPrefsWritable vectorAndPrefsWritable,
                  Context context) throws IOException, InterruptedException {

    Vector cooccurrenceColumn = vectorAndPrefsWritable.getVector();
    List<Long> userIDs = vectorAndPrefsWritable.getUserIDs();
    List<Float> prefValues = vectorAndPrefsWritable.getValues();

    for (int i = 0; i < userIDs.size(); i++) {
      long userID = userIDs.get(i);
      float prefValue = prefValues.get(i);
```

```
      Vector partialProduct = cooccurrenceColumn.times(prefValue);
      context.write(new VarLongWritable(userID),
                    new VectorWritable(partialProduct));
    }
  }
}
```

This mapper writes a very large amount of data. For each user-item association, it outputs one copy of an entire column of the co-occurrence matrix. That's more or less necessary; those copies must be grouped together with copies of other columns in a reducer, and summed, to produce the recommendation vector.

But one optimization comes into play at this stage: a combiner. This is like a miniature reducer operation (in fact, combiners extend `Reducer` as well) that is run on map output while it's still in memory, in an attempt to combine several of the output records into one before writing them out. This saves I/O, and it isn't always possible. Here, it is; when outputting two vectors, *A* and *B*, for a user, it's just as good to output one vector, *A* + *B*, for that user—they're all going to be summed in the end.

Listing 6.8 Combiner for partial products

```
public class AggregateCombiner extends
    Reducer<VarLongWritable,VectorWritable,
            VarLongWritable,VectorWritable> {
  public void reduce(VarLongWritable key,
                     Iterable<VectorWritable> values,
                     Context context)
      throws IOException, InterruptedException {
    Vector partial = null;
    for (VectorWritable vectorWritable : values) {
      partial = partial == null ?
        vectorWritable.get() : partial.plus(vectorWritable.get());
    }
    context.write(key, new VectorWritable(partial));
  }
}
```

Listing 6.8 shows such a combiner, which operates on the output of `PartialMultiplyMapper`. How much I/O it'll save depends on how much output Hadoop collects in memory before writing to disk (the map spill buffer), and how frequently columns for one user show up in the output of one map worker. That is, if a lot of `Mapper` output is collected in memory, and many of those outputs can be combined, then the combiner will save a lot of I/O. Mahout's implementation of these jobs will attempt to reserve much more memory than usual for the mapper output by increasing the value of `io.sort.mb` up to a gigabyte.

6.3.6 Translating to MapReduce: making recommendations

At last, the pieces of the recommendation vector must be assembled for each user so that the algorithm can make recommendations. The following listing shows this in action.

Listing 6.9 Producing recommendations from vectors

```
public class AggregateAndRecommendReducer extends
    Reducer<VarLongWritable,VectorWritable,
            VarLongWritable,RecommendedItemsWritable> {

  ...

  public void reduce(VarLongWritable key,
                     Iterable<VectorWritable> values,
                     Context context)
      throws IOException, InterruptedException {

    Vector recommendationVector = null;
    for (VectorWritable vectorWritable : values) {
      recommendationVector = recommendationVector == null ?
        vectorWritable.get() :

     recommendationVector.plus(vectorWritable.get());
    }

    Queue<RecommendedItem> topItems = new PriorityQueue<RecommendedItem>(
          recommendationsPerUser + 1,
          Collections.reverseOrder(
            ByValueRecommendedItemComparator.getInstance()));

    Iterator<Vector.Element> recommendationVectorIterator =
        recommendationVector.iterateNonZero();
    while (recommendationVectorIterator.hasNext()) {
      Vector.Element element = recommendationVectorIterator.next();
      int index = element.index();
      float value = (float) element.get();
      if (topItems.size() < recommendationsPerUser) {
        topItems.add(new GenericRecommendedItem(
                          indexItemIDMap.get(index), value));
      } else if (value > topItems.peek().getValue()) {
        topItems.add(new GenericRecommendedItem(
                          indexItemIDMap.get(index), value));
        topItems.poll();
      }
    }

    List<RecommendedItem> recommendations =
      new ArrayList<RecommendedItem>(topItems.size());
    recommendations.addAll(topItems);
    Collections.sort(recommendations,
                     ByValueRecommendedItemComparator.getInstance());
    context.write(
        key,
        new
     RecommendedItemsWritable(recommendations));
  }
}
```

Build recommendation vector by summing

Find top *N* highest values

Output recommendations in order

The output ultimately exists as one or more files stored on an HDFS instance, as a compressed text file. The lines in the text file are of this form:

```
3    [103:24.5,102:18.5,106:16.5]
```

Each user ID is followed by a comma-delimited list of item IDs that have been recommended (followed by a colon and the corresponding entry in the recommendation vector, for what it's worth). This output can be retrieved from HDFS, parsed, and used in an application. Note that the output from Mahout will be compressed using gzip, to save space.

At last, we've ended up with recommendations, having started with raw input from the Wikipedia data set. As you can see, even this simple recommender implementation looks quite different when designed and implemented on a distributed system like Hadoop. Even in simplified form, it consists of five stages, each performing a piece of the computation or transformation along the way. The natural next step is to run all of this, to better understand how such an implementation is put to work in practice.

6.4 Running MapReduces with Hadoop

Now it's time to try out this implementation on the Wikipedia links data set. Although Hadoop is a framework for running a computation across clusters of potentially thousands of machines, you'll start by running a Hadoop computation on a cluster of one machine: yours. It's far simpler to set up a Hadoop cluster on your local machine for learning than to set up a proper cluster.

This process won't be very different from using a real cluster. First, you must set up a pseudo-distributed Hadoop cluster on your local machine, and then you'll be able to experiment with running these and other MapReduces in Mahout.

6.4.1 Setting up Hadoop

As mentioned in chapter 1, you'll need to download a recent of copy of Hadoop from Apache (http://hadoop.apache.org/common/releases.html). Version 0.20.2 is the recommended version at the time of writing. Follow the setup directions at http://hadoop.apache.org/common/docs/current/single_node_setup.html and configure for what it calls pseudo-distributed operation.

Before running the Hadoop daemons with `bin/start-all.sh`, make one additional change: in conf/mapred-site.xml, add a new property named `mapred.child.java.opts` with the value `-Xmx1024m`. This will enable Hadoop workers to use up to 1 GB of heap memory. You can stop following the setup instructions after you've started running all the Hadoop daemons.

You're now running a complete Hadoop cluster on your local machine, including an HDFS instance. You need to put the input onto HDFS to make it available to Hadoop. You may wonder why, if the data is readily available on the local filesystem, it needs to be copied again into HDFS. Recall that, in general, Hadoop is a framework run across many machines, so any data it uses needs to be available not just to one machine but many. HDFS is an entity that can make data available to these many machines. Copy the input to HDFS with this command:

```
bin/hadoop fs -put links-simple-sorted.txt input/input.txt
```

Computing recommendations for *every* article in the data set would take a long time, because you're running on only one machine and incurring all the overhead of the distributed computing framework. You can ask the implementation in Mahout to compute recommendations for, say, just one user. Create a file containing only the number 3 on a single line, and save it as users.txt. This is a list of the articles for which the algorithm will generate recommendations—here, one, for testing purposes. Place in into the HDFS instance as well, with this command:

```
bin/hadoop fs -put users.txt input/users.txt
```

6.4.2 Running recommendations with Hadoop

The glue that binds together the various `Mapper` and `Reducer` components is `org.apache.mahout.cf.taste.hadoop.item.RecommenderJob`. It can be found within the Mahout source distribution. It configures and invokes the series of MapReduce jobs discussed previously. These MapReduces and their relationships are illustrated in figure 6.2.

In order to run `RecommenderJob`, and allow Hadoop to run these jobs, you need to combine all of this code into one JAR file, along with all of the code it depends upon. This can be accomplished easily by running `mvn clean package` from the core/ directory in the Mahout distribution—this will produce a file like mahout-core-0.5-job.jar. Or you can use a precompiled job JAR from Mahout's distribution.

Now, kick it all off with this command:

```
hadoop jar mahout-core-0.5-job.jar \
  org.apache.mahout.cf.taste.hadoop.item.RecommenderJob \
  -Dmapred.input.dir=input/input.txt \
  -Dmapred.output.dir=output --usersFile input/users.txt --booleanData
```

Hadoop will take over and begin running the series of jobs. It will take many hours, because only one machine (yours) is being deployed to complete the computations. Even with a small army of machines, don't expect results in minutes; the overhead of initializing the cluster, distributing the data and executable code, and marshaling the results is nontrivial.

In fact, running recommendations for all 5.7 million or so users in the data set took almost 700 hours of CPU core time on a local one-machine cluster, or about half a second per recommendation. Running time is dominated by the final phase, where vectors are multiplied and added. That's more time than required by the nondistributed recommenders at somewhat smaller scales, but not orders of magnitude worse. At current prices for virtual instance time on services like Amazon Elastic MapReduce, this would cost about $0.01 per 1,000 users' worth of recommendations. Of course, it need not take 700 hours to compute all these recommendations; if you're employing 100 worker machines, the entire process would take somewhat more than 7 hours of actual time to complete.

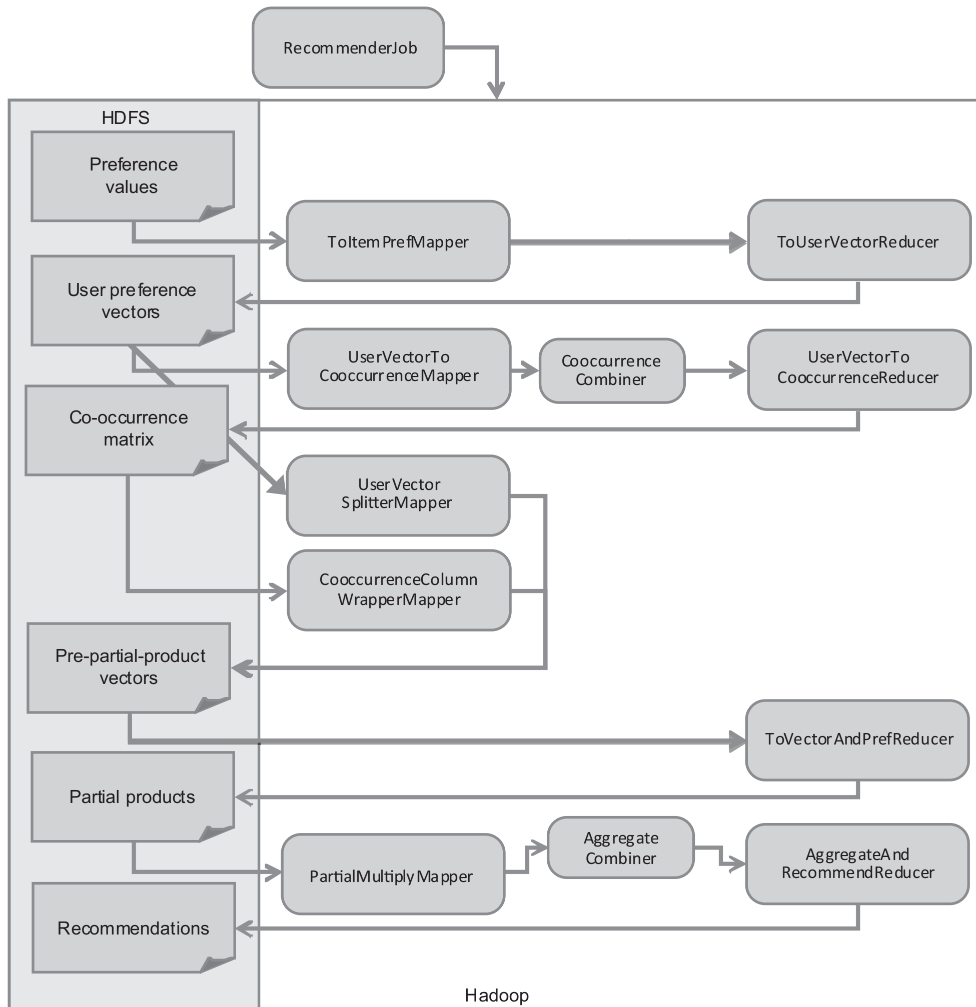


Figure 6.2 The relationships between `RecommenderJob`, the `MapReduces` it invokes, and the data that they read to and write from HDFS

Back on your local instance, if you're patient enough to let it complete the recommendations for just one user, you should find the results on HDFS under the output/ directory. It will be contained in a single file called part-r-00000.

Copy the results back to your local filesystem with `bin/hadoop fs -get output/part-r-00000`. This file can be examined and used as desired.

Congratulations, that's it! You've produced recommendations with a fully distributed framework (on a cluster of one machine). Don't forget to shut down Hadoop with `bin/stop-all.sh` when you're done.

6.4.3 Configuring mappers and reducers

One important point deserves mention here. Previously, you let Hadoop default to running just one map and one reduce worker at once. This was appropriate, because the algorithm was running on just one machine. In general, though, when launching this job on a cluster of many machines, one worker is too little. On a real cluster, the number of workers can be controlled with command-line arguments like

```
-Dmapred.map.tasks=X -Dmapred.reduce.tasks=Y
```

Setting *X* and *Y* equal to the total number of cores available in the cluster is a good place to start. For example, if your cluster has five quad-core machines, set both to 20.

You've now successfully designed and implemented a distributed algorithm on Hadoop, set up a Hadoop cluster, and run your implementation on Hadoop. The good news is that this constitutes most of the learning curve; interacting with a real Hadoop cluster is virtually the same. You're now well prepared for encounters with a number of other MapReduce-based and Hadoop-based machine learning algorithms in Mahout, which will be presented in coming chapters.

Before closing out our coverage of recommenders, it's useful to examine one more Hadoop-based recommender algorithm offered by Mahout, which is a sort of hybrid between the distributed and nondistributed approaches you've seen so far.

6.5 Pseudo-distributing a recommender

Earlier, you saw how to create, test, and operate a variety of nondistributed recommender engines with Mahout, on one machine. This chapter demonstrated how to run one fully distributed recommender computation using a quite different approach. There's a middle ground, however, for applications that want to use multiple machines but also want to use an existing nondistributed implementation.

This might be the case for applications that have already developed an effective customized implementation using the nondistributed framework. Such a `Recommender` implementation is probably, like all the nondistributed implementations, intimately bound to a `DataModel` to do its work, and it assumes efficient, random access to all available data. It might be hard or impossible to reinvent it in a fully distributed form.

For these situations, Mahout offers a pseudo-distributed recommender engine framework. This is merely a Hadoop-based harness that can run several independent, nondistributed instances of a given recommender engine in parallel. As such, it's an easy way to allow a stock, nondistributed algorithm to use many machines. This facility doesn't actually parallelize the computation in any sense; it only manages the operation of multiple nondistributed instances. Performance is the same as when running a nondistributed instance directly, but this allows you to run n instances of the recommender on n machines, each producing $1/n$ of all the recommendations required, in a total of $1/n$ the time it would take one machine to finish.

The disadvantage to this approach is a scalability limitation: a nondistributed computation remains limited in the amount of data it can handle by the resources of the

machines that it runs on—a computation that can't fit on one large machine still won't fit when sent to *n* independent large machines. Pseudo-distributing the computation doesn't change this.

There are no new algorithms or code to introduce here; the pseudo-distributed recommender engine framework in Mahout runs the existing `Recommender`s, but on Hadoop. Conceptually, it uses Hadoop to split the set of users across *n* machines, copy the input data to each, and then run one `Recommender` on each machine to process recommendations for a subset of users.

The process is the same as before. With Hadoop set up and running, copy the preferences input file into HDFS. If you wish to try out this framework, choose an input such as ua.base from the GroupLens 100K data set. (The Wikipedia links data set will be too large to use with a nondistributed implementation.) The input file within this data set, ua.base, needs one small conversion to work with this code as it is now. The tabs must be converted to commas. You can do this in your favorite editor, or with a Unix command line like this:

```
cut -f1-3 ua.base | tr '\t' ',' > ua.base.hadoop
```

Place ua.base.hadoop into HDFS under, for example, input/ua.base.hadoop.

The framework needs the name of a `Recommender` implementation that it can instantiate and use. The only requirement is that the implementation must provide a constructor that takes a single argument, a `DataModel`. With this, the framework can do the rest. Typically, you would supply a customized `Recommender` that you had created for your application here. For testing purposes, though, `SlopeOneRecommender` will do, because it can be instantiated with only a `DataModel` as configuration.

The mahout-core-0.5-job.jar file already contains `SlopeOneRecommender`, because it's a standard Mahout implementation. But were you to use your own implementation, you would need to add it and any of its dependent classes into the JAR file as well. This can be accomplished with

```
jar uf mahout-core-0.5-job.jar -C [classes directory]
```

where the classes directory is the location where your IDE or build tool output the compiled version of your code.

Finally, run the job:

```
hadoop jar mahout-core-0.5-job.jar \
  org.apache.mahout.cf.taste.hadoop.pseudo.RecommenderJob \
  -Dmapred.input.dir=input/ua.base.hadoop \
  -Dmapred.output.dir=output \
  --recommenderClassName \
  org.apache.mahout.cf.taste.impl.recommender.slopeone.SlopeOneRecommender
```

As before, you'll find the output in HDFS in the output/ directory.

That's all there is to it; if your input is of such a scale that truly distributed algorithms aren't required, the pseudo-distributed recommender framework is a quick and easy way to utilize more computing power to produce recommendations faster.

6.6 Looking beyond first steps with recommendations

The distributed portion of Mahout's recommendation engine introduced in this chapter is still under construction, so refer to the latest documentation and code in conjunction with this book. Look to Mahout to provide more options as it evolves. It's a work in progress from a growing community; we aim to continually improve and expand our understanding of and ability to deploy recommenders and machine learning.

Before moving on from recommenders to clustering and classifier algorithms, we conclude with a collection of brief, parting thoughts about where to go next from here in your thinking and investigation of recommender engines.

6.6.1 Running in the cloud

Don't have a hundred machines lying around on which to run these big distributed computations? Fortunately, today, service providers allow you to rent storage and computing time from a computing cloud.

Amazon's Elastic MapReduce service (http://aws.amazon.com/elasticmapreduce/) is one such service, pictured in figure 6.3. It uses Amazon's S3 storage service instead of a pure HDFS instance for storing data in the cloud. After uploading your JAR file and data to S3, you can invoke a distributed computation using their AWS Console by supplying the same arguments you used to invoke the computation on the command line earlier.

DEFINE JOB FLOW · SPECIFY PARAMETERS · CONFIGURE EC2 INSTANCES · BOOTSTRAP ACTION

Please review the details of your job flow and click "Create Job Flow" when you are ready to l

Job Flow Name: My Job Flow
Type: Custom Jar

Jar Location: s3://my-bucket/mahout.jar
Jar Arguments: org.apache.mahout.cf.taste.hadoop.item.RecommenderJob --Dmapred.map.tasks=10 -Dmapred.reduce.tasks=10 --

Number of Instances: 10
Type of Instance: m1.small
Amazon EC2 Key Pair:
Amazon S3 Log Path:
Enable Hadoop Debugging: No

Bootstrap Actions: No Bootstrap Actions created for this Job Flow

‹ Back · Create Job Flow

Figure 6.3 Amazon's AWS Elastic MapReduce console

After logging in to the main AWS Console, select the Amazon Elastic MapReduce tab. Choose the Create New Job Flow option. Give the new flow whatever name you like, and specify Run Your Own Application. Choose the Custom Jar type and continue. Specify the location on S3 where the JAR file resides; this will be an s3: URI, not unlike s3://my-bucket/target/mahout-core-0.5-job.jar.

The job arguments will be the same as when running on the command line; here it'll certainly be necessary to configure the number of mappers and reducers. The number of mappers and reducers can be tuned to your liking; as earlier, start with a number equal to the number of virtual cores you reserve for the computation. Although any instance type can be used, start with the regular types unless there is a reason to choose something else: small, large, or extra-large. The number and type of instances is selected on the next AWS Console screen.

If your input data is extremely large, some recommender jobs such as that in `org.apache.mahout.cf.taste.hadoop.item` may need more RAM per mapper or reducer. In this case, you may have to choose a high-memory instance type. You may also opt for a high-CPU instance type; the risk is that the jobs will spend so much time reading and writing data to S3 that these instances' speedy CPUs will go mostly unused. Therefore, the conventional instance types are a good place to start. If you're using the small instance type, which has one virtual core per instance, set the number of mappers and reducers equal to the number of instances you select.

You can leave the other options untouched unless you have a reason to set them.

This is the essence of running a recommender job on Elastic MapReduce; refer to Amazon's documentation for more information about how to monitor, stop, and debug such jobs.

6.6.2 Imagining unconventional uses of recommendations

Our discussion of recommenders is drawing to a close. Before moving on to clustering, though, it's worth changing gears to provide some parting thoughts on how recommenders might apply to your projects.

Although the Mahout recommender engine APIs are phrased in terms of users and items, the framework doesn't assume that users are people and items are objects like books and DVDs. You already applied recommender engines to a dating site's data to recommend *people* to *people*, for example. What other ways can recommender engines be applied? Here are some ideas to fire your imagination:

- *Recommend users to items*—By swapping item IDs for user IDs, a recommender engine's output instead suggests which *users* might be most interested in a given *item*.
- *Think broadly about items*—Given associations from users to places, times, usage patterns, or other people, you can recommend the same back to them.
- *Find most-similar items*—Item-based recommender implementations in Mahout make it easy to find a set of most-similar items, which could be useful to present to users as well.

- *Think broadly about preference values*—It's unusual to be able to collect explicit preference values from users. Think about what you can *infer* from the data you do have about users' relations to things.
- *Think about more than just one user and item*—You can recommend to pairs of users by thinking of a pair of users as a user. You can recommend, say, an item and place by taking both of them together as an item.

Mahout doesn't offer particular, special support for these use cases, though all of them can be implemented on top of Mahout. This is a possible direction in which Mahout could grow, or in which specialized third-party projects might appear. In particular, the problem of inferring implicit ratings based on user behavior and other data is a fascinating and important problem in its own right, but not one that Mahout addresses.

6.7 Summary

In this chapter, we took a brief look at the large data set based on Wikipedia article links. With 130 million preferences, it's large enough to require a different, distributed approach to produce recommendations.

We discussed the tradeoffs inherent in moving from one machine and a nondistributed algorithm to a large distributed computation on a cluster of machines. Then we briefly introduced the MapReduce paradigm and its implementation in Hadoop as a way to manage such distributed computations.

We translated the item-based recommender algorithm we saw before into a different distributed implementation that relies on matrix and vector operations to discover the best recommendations. We returned to the Wikipedia data set, prepared it for use with Hadoop, and walked through creating recommendations for this data set on a local Hadoop and HDFS instance.

Finally, we examined pseudo-distributed recommender computations with Mahout: running several independent instances of nondistributed `Recommender` implementations on Hadoop.

This concludes our coverage of recommender engines in Mahout. It has been intended as a gentle introduction to one aspect of machine learning, which gradually evolved from small input and nondistributed computations to large-scale distributed computations. Now, we'll move on to discuss clustering and classification with Mahout, which entails more complex machine learning theory and more intense use of distributed computing. With recommender engines under your belt, you're ready to engage these topics. Read on.

Part 2

Clustering

This part of the book, including chapters 7 through 12, explores clustering algorithms in Apache Mahout. With the techniques described here, you can group together similar-looking pieces of data into a set or a cluster. Clustering helps uncover interesting groups of information in a large volume of data. This part of the book begins with simple problems in clustering involving examples written in Java. As we progress, you'll see more real-world examples and learn how to make Apache Mahout run as Hadoop jobs that can cluster large data easily.

Chapter 7 introduces the notion of clustering and explains it with an example of clustering points in a 2-dimensional plane. Chapter 8 introduces the concept of vectors and explains how data can be represented using them. Chapter 9 introduces various clustering algorithms implemented in Apache Mahout. With clear examples, this chapter shows you how each clustering algorithm fits various use cases.

Chapter 10 shows how clustering quality can be measured and how it can be improved by tweaking various options and parameters in Mahout. Chapter 11 explores the distributed side of the Apache Mahout clustering implementations and explains how clustering can be run as Hadoop jobs on large data sets. Finally, chapter 12 uses what you've learned from all the other chapters, explores some real-world problems in clustering, and solves them using Apache Mahout.

7 Introduction to clustering

This chapter covers

- A hands-on look at clustering
- Understanding the notion of similarity
- Running a simple clustering example in Mahout
- The various distance measures used for clustering

As human beings, we tend to associate with like-minded people—"birds of a feather flock together." We have a great mental ability for finding repeating patterns, and we continually associate what we see, hear, smell, and taste with things that are already in our memory. For example, the taste of honey reminds us more of the taste of sugar than salt. So we group together the things that taste like sugar and honey and call them sweet. Without even knowing what sweet tastes like, we know that all the sugary things in the world are similar and of the same group. We also know how different they are from all the things belonging to the salty group. Unconsciously, we group together tastes into such *clusters*. We have clusters of sugary things and salty things, with each group having hundreds of items in it.

In nature, we observe many other types of groups. Consider apes versus monkeys, which are both kinds of primates. All monkeys share some traits like a shorter height, a long tail, and a flat nose. On the other hand, apes have a larger size, long

arms, and bigger heads. Apes look different from monkeys, but both are fond of bananas. So we can think of apes and monkeys as two different groups or as a single group of banana-loving primates. What we consider as a cluster depends entirely on the traits we choose for measuring the similarity between items (in this case, primates).

In this chapter, you'll learn what clustering is, how it relates to concepts in mathematics, and see an example of clustering using Mahout. Let's start with the basics.

7.1 Clustering basics

So what is the process of clustering all about? Suppose you were given the keys to a library containing thousands of books. The books were arranged in no particular order. Readers entering your library would have to sweep through all the books, one by one, to find a particular book. Not only is this cumbersome and slow, it's tedious as well.

Sorting the books alphabetically by title would be a vast improvement—for readers searching for a book by title, that is. What if most people were simply browsing, or researching a general subject? Grouping the books by topic would be more useful than an alphabetical-by-title ordering. But how would you even begin this grouping? Having just taken over this job, you wouldn't even be sure what all the books were about—surfing, romance, or topics you hadn't encountered before.

To group the books by topic, you could lay them all down in a line and start reading them one by one. When you encountered a book whose content is similar to a previous book, you could go back and stack them together. When you finished, you'd have hundreds of stacks of books instead of thousands of individual books.

Good work! This was your first clustering experience. If a hundred topic groups are too many, you could go back to the beginning of the line and repeat the process with the stacks, until your stacks were on topics quite different from one another.

Clustering is all about organizing items from a given collection into groups of similar items. These clusters could be thought of as sets of items similar to each other in some ways but dissimilar from the items belonging to other clusters.

Clustering a collection involves three things:

- *An algorithm*—This is the method used to group the books together.
- *A notion of both similarity and dissimilarity*—In the previous discussion, we relied on your assessment of which books belonged in an existing stack and which should start a new one.
- *A stopping condition*—In the library example, this might be the point beyond which books can't be stacked anymore, or when the stacks are already quite dissimilar.

Until now, we have thought of clustering items as stacking them, but really you were just grouping them. Conceptually, clustering is more like looking at which items form near groups and circling them. Figure 7.1 shows the clustering of points in a standard x-y plane. Each circle represents one cluster containing several points.

In this simple example, the circles obviously represent the best clustering of points into three clusters based on distance. Circles are good way to think of clusters,

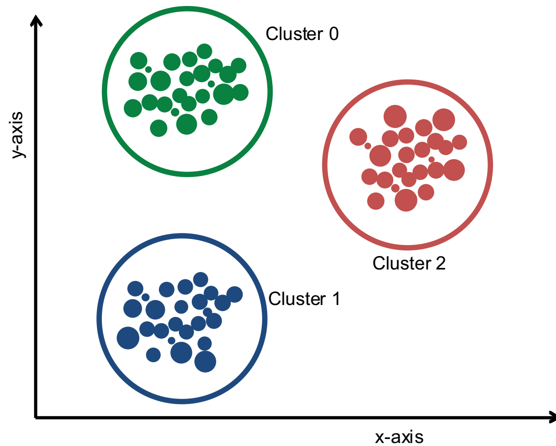


Figure 7.1 Points in an x-y plane. The circles represent clusters, and the points on the plane could be viewed as three logical groups. Clustering algorithms can help identify those groups.

because clusters are also defined by a center point and radius. The center of the circle is called the *centroid*, or *mean* (*average*), of that cluster. It's the point whose coordinates are the average of the x and y coordinates of all points in the cluster.

In this chapter, we visualize clustering as a geometric problem. We also use techniques from geometry to explain the various distance measures, which are core to clustering algorithms. We also look at some of the important distance measures and their relationships with clustering. This will help you visualize the acute similarity between clustering points in a plane and clustering text documents.

In later chapters, we explore some of the methods that are popularly used for clustering data, and the way they're implemented in software in Mahout. The strategy in the library examples was to merge stacks of books until some threshold was reached. The number of clusters formed in this example depended on the data; based on the number of books and the threshold, you might have ended up with 100, 20, or even just 1 cluster. A more popular strategy is to set a target number of clusters, instead of a threshold, and then find the best grouping with that constraint. Later we explain this and other variations in detail.

7.2 Measuring the similarity of items

The most important issue in clustering is finding a function that quantifies the similarity between any two data points as a number. Note that we're using the terms *item* and *point* interchangeably in this whole book. Both refer to a unit of data that's to be clustered.

In the x-y plane example, the measure of similarity (the *similarity metric*) for the points was the Euclidean distance between two points. The library example had no such clear, mathematical measure and instead relied entirely on the wisdom of the librarian to judge the similarity between books. That surely doesn't work in our case, because we need a metric that can be implemented on a computer.

One possible metric could be based on the number of words common to two books' titles. Based on that *Harry Potter: The Philosopher's Stone* and *Harry Potter: The*

Prisoner of Azkaban have three words in common: *Harry*, *Potter*, and *The*. Even though *The Lord of the Rings: The Two Towers* is similar to the Harry Potter series, this measure of similarity doesn't capture that at all. You'd need to alter the similarity metric to take account of the contents of the books themselves. You could assemble word counts for each book, and when the counts are close for many overlapping words, you could judge the books to be similar.

Unfortunately, that's easier said than done. Not only do these books have hundreds of pages of text, but features of English also confound this sort of measure. The most frequent words in these English-language texts are words like *a*, *an*, and *the*, which invariably occur frequently in both books but say little about their similarity.

To combat this effect, you could use numeric weights in the computation, and apply low weights to these words to reduce their effect on the similarity value. You should give less weight to words that occur across many books, and more weight to words that are found in few books. You could also weight words occurring more often in a particular book because those words strongly suggest the content of the books—like *magic* in the case of Harry Potter.

Once you give a weight value to each word in a book, you can say that the similarity between two books is the sum over all words—the frequency of a particular word in the two books times their weighting. This is a decent measure, if the books are of equal length.

What if one book is 300 pages long and the other is 1,000 pages long? Surely, the larger book will have a larger word count in general. You'll have to ensure that the weight of words is relative to the length of the text. Weighting words in free-form text is a science of its own.

These and many other tricks are part of a popular weighting method called TF-IDF (term frequency–inverse document frequency). We start with points on a plane in this chapter and progress toward understanding weighting methods in the next couple of chapters. We go in depth into TF-IDF and its impact on quality of clustering. Let's begin with a Hello World clustering example.

7.3 Hello World: running a simple clustering example

Mahout contains various implementations of clustering, like *k-means*, *fuzzy k-means*, and *canopy* to name a few. For our Hello World example, you'll run the k-means algorithm. In upcoming chapters, we review other clustering algorithms in Mahout and their real-world applications.

7.3.1 Creating the input

First, let's try a simple example that clusters points in two dimensions, like the illustration you saw in figure 7.1.

We start by creating a list of points to cluster. Mahout clustering algorithms take input in a particular binary format called `SequenceFile`, a format used extensively by the Hadoop library. The input encodes `Vectors`, each of which represents one point.

Listing 7.1 Sample input for our first clustering example

```
(1,1)
(2,1)
(1,2)
(2,2)
(3,3)
(8,8)
(8,9)
(9,8)
(9,9)
```

Examine the sample input in listing 7.1. Figure 7.2 plots it on the x-y plane, and two clusters stand out clearly. One cluster contains five points in one region of the plane, and the other contains four points in another region.

There are three steps involved in inputting data for the Mahout clustering algorithms: you need to preprocess the data, use that data to create vectors, and save the vectors in `SequenceFile` format as input for the algorithm. In the case of points, no preprocessing is necessary because they're already vectors in the 2-dimensional plane—you just need to convert them to a Mahout `Vector` class.

When you hear the word vector, you may be recalling your high school physics course, where vectors were arrows with directions, and not single points in space. For our purposes in machine learning, the term vector refers to an ordered list of numbers, which is all a point or physics vector is anyway. `Vectors` have a number of dimensions (here, two dimensions) and a numeric value for each dimension.

> **TIP** Appendix B explains the `Vector` interface and its implementations in some detail; refer to it as needed to better understand how Mahout represents vectors. The details aren't yet critical to our example.

In the following subsection, we'll cluster 2-dimensional points using Mahout. The `getPoints` function will be used to convert the given set of input points to `RandomAccessSparseVector` format. Once these vectors are generated, they're written in the

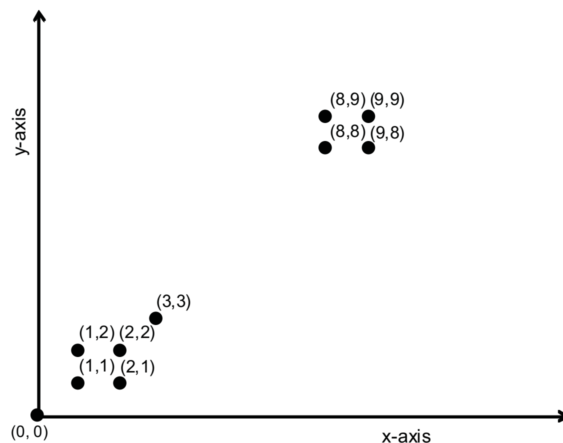


Figure 7.2 Plot of input points from listing 7.1 in the x-y plane

SequenceFile format that the clustering algorithms in Mahout can read. The write-PointsToFile function shows how this is done.

7.3.2 *Using Mahout clustering*

Once the input is ready, you can cluster the points. In this example, we use the k-means clustering algorithm, which takes the following input parameters:

- The SequenceFile containing the input vectors.
- The SequenceFile containing the initial cluster centers. In this case, we seed two clusters, so there are two centers.
- The similarity measure to be used. We use EuclideanDistanceMeasure as the measure of similarity here, and we explore other similarity measures later in this chapter.
- The convergenceThreshold. If in a particular iteration the centers of the clusters don't change beyond this threshold, no further iterations are done.
- The number of iterations to be done.
- The Vector implementation used in the input files.

You have everything you need, except the initial set of cluster centers. Because you're trying to generate two clusters from the nine points, you need two points in the initial set of centers, as shown in figure 7.3. This initial set of two points serves as the best guess of the two cluster centers you're trying to find using the k-means algorithm.

Of course, you can observe that these guesses aren't very good; both clearly fall within one of the apparent clusters. Unfortunately, in nontrivial examples, there is no way to know beforehand where the clusters are. There are various methods to estimate the centers of the clusters—the canopy clustering algorithm can do this estimation quickly and efficiently.

Even if the estimated centers are way off, the k-means algorithm will readjust them at the end of each iteration by computing the average center, or the *centroid*, of all

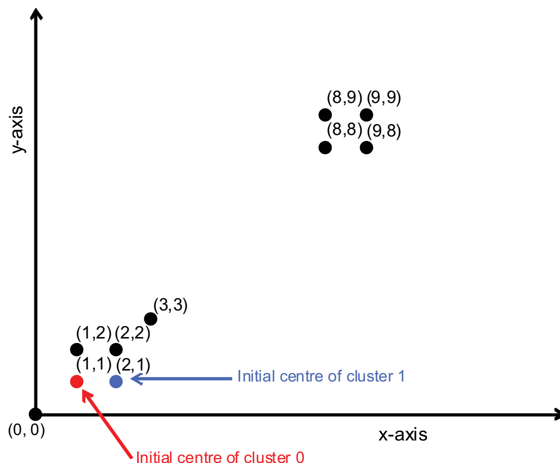


Figure 7.3 Marking the initial clusters is an important step in k-means clustering.

points in the cluster. To demonstrate this corrective nature of k-means, you'll start with center points taken close together—(1, 1) and (2, 1). You can also play with different values for the input center points and see how k-means allows various combinations of centers to converge.

The following listing shows the code, which uses Mahout k-means in the in-memory mode to cluster the set of points on the plane.

Listing 7.2 The Hello World clustering code

```
public static final double[][] points = { {1, 1}, {2, 1}, {1, 2},
                                          {2, 2}, {3, 3}, {8, 8},
                                          {9, 8}, {8, 9}, {9, 9}};

public static void writePointsToFile(List<Vector> points,
                                  String fileName,
                                  FileSystem fs,
                                  Configuration conf) throws IOException {
  Path path = new Path(fileName);
  SequenceFile.Writer writer = new SequenceFile.Writer(fs, conf,
      path, LongWritable.class, VectorWritable.class);
  long recNum = 0;
  VectorWritable vec = new VectorWritable();
  for (Vector point : points) {
    vec.set(point);
    writer.append(new LongWritable(recNum++), vec);
  }
  writer.close();
}

public static List<Vector> getPoints(double[][] raw) {
  List<Vector> points = new ArrayList<Vector>();
  for (int i = 0; i < raw.length; i++) {
    double[] fr = raw[i];
    Vector vec = new RandomAccessSparseVector(fr.length);
    vec.assign(fr);
    points.add(vec);
  }
  return points;
}

public static void main(String args[]) throws Exception {

  int k = 2;                                        Specify number of clusters to be formed

  List<Vector> vectors = getPoints(points);         Create input directories for data

  File testData = new File("testdata");
  if (!testData.exists()) {
    testData.mkdir();
  }
  testData = new File("testdata/points");
  if (!testData.exists()) {
    testData.mkdir();
  }
```

```
  Configuration conf = new Configuration();
  FileSystem fs = FileSystem.get(conf);
  writePointsToFile(vectors,
      "testdata/points/file1", fs, conf);                  <- Write initial centers

  Path path = new Path("testdata/clusters/part-00000");
  SequenceFile.Writer writer
      = new SequenceFile.Writer(
          fs, conf,       path, Text.class, Cluster.class);

  for (int i = 0; i < k; i++) {
    Vector vec = vectors.get(i);
    Cluster cluster = new Cluster(
        vec, i, new EuclideanDistanceMeasure());
    writer.append(new Text(cluster.getIdentifier()), cluster);
  }
  writer.close();

  KMeansDriver.run(conf, new Path("testdata/points"),       <- Run k-means
    new Path("testdata/clusters"),                             algorithm
    new Path("output"), new EuclideanDistanceMeasure(),
        0.001, 10, true, false);

  SequenceFile.Reader reader
      = new SequenceFile.Reader(fs,
          new Path("output/" + Cluster.CLUSTERED_POINTS_DIR
              + "/part-m-00000"), conf);

  IntWritable key = new IntWritable();
  WeightedVectorWritable value = new WeightedVectorWritable();
  while (reader.next(key, value)) {
    System.out.println(
        value.toString() + " belongs to cluster "          <- Read output,
          + key.toString());                                   print vector,
  }                                                            cluster ID
  reader.close();
}
```

To get a clear picture of what listing 7.2 does, take a look at the flow in figure 7.4.

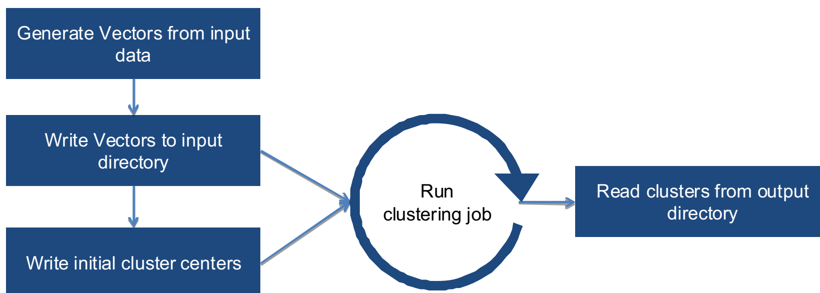


Figure 7.4 Flow of the Hello World clustering example

7.3.3 Analyzing the output

Compile and run the code in listing 7.2 using your favorite IDE or do it from the command line. Make sure you add all the Mahout dependency JAR files to the classpath. Because our set of data is small, you'll get the following output in a matter of seconds:

```
1.0: [1.000, 1.000] belongs to cluster 0
1.0: [2.000, 1.000] belongs to cluster 0
1.0: [1.000, 2.000] belongs to cluster 0
1.0: [2.000, 2.000] belongs to cluster 0
1.0: [3.000, 3.000] belongs to cluster 0
1.0: [8.000, 8.000] belongs to cluster 1
1.0: [9.000, 8.000] belongs to cluster 1
1.0: [8.000, 9.000] belongs to cluster 1
1.0: [9.000, 9.000] belongs to cluster 1
```

The code in listing 7.2 marks each vector with a string identifier to uniquely identify it. This helps you to evaluate and reconstruct the clusters later. As you can see in figure 7.5, the algorithm was able to readjust the center of the cluster 1 from (2, 1) to (8.5, 8.5)—the centroid of all points in cluster 1.

In this simple example, Mahout clustered the points into two sets quickly and with good accuracy. Real-world data isn't as simple. With millions of input vectors, each having millions of dimensions, clustering quickly becomes nontrivial, and quality and performance issues arise. It'll be difficult to decide how many clusters to produce, or what kind of similarity measures to choose. Tuning the performance and even evaluating the quality of the clusters will require some effort. Getting the perfect clustering is a never-ending task.

7.4 Exploring distance measures

Mahout clustering implementations are configurable enough to fit the needs of almost any clustering problem—the question is which configuration is best! One of the key factors in the configuration is the choice of distance measure.

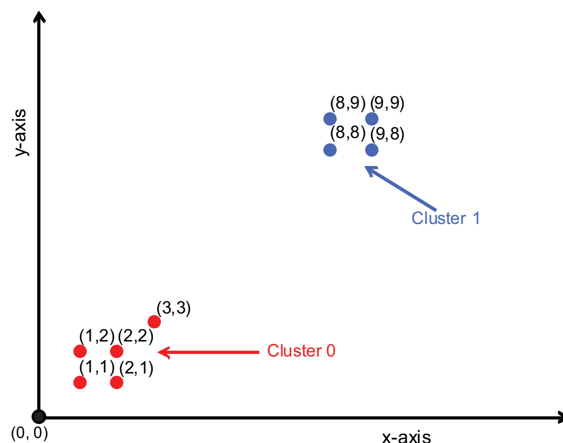


Figure 7.5 The output of the Hello World k-means clustering program. Even with distant centers, the k-means algorithm was able to correctly iterate and correct the centers based on the Euclidean distance measure.

In the previous example, we used `EuclideanDistanceMeasure` to calculate the distance between points. Although it proved to be an effective measure in generating the clusters, there are other similarity measure implementations in the Mahout clustering package. Aptly named `DistanceMeasure` implementations, these classes calculate the distance between two vectors according to various definitions of distance. Shorter distances indicate more similarity between the vectors and vice versa; similarity and distance are related concepts.

7.4.1 Euclidean distance measure

The Euclidean distance, which you've already seen, is the simplest of all distance measures. It's the most intuitive and matches our normal idea of distance.

For example, given two points on a plane, the Euclidean distance measure could be calculated by using a ruler to measure the distance between them. Mathematically, Euclidean distance between two *n*-dimensional vectors $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_n)$ is



$$d = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_n - b_n)^2}$$

No. 7

The Mahout class that implements this measure is `EuclideanDistanceMeasure`.

7.4.2 Squared Euclidean distance measure

Just as the name suggests, this distance measure's value is the square of the value returned by the Euclidean distance measure. For *n*-dimensional vectors $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_n)$ the distance becomes

$$d = (a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_n - b_n)^2$$

The Mahout class that implements this measure is `SquaredEuclideanDistanceMeasure`.

7.4.3 Manhattan distance measure

Unlike Euclidean distance, under the Manhattan distance measure, the distance between any two points is the sum of the absolute differences of their coordinates. Figure 7.6 compares the Euclidean distance and Manhattan distance between two points in the x-y plane.

This distance measure takes its name from the grid-like layout of streets in Manhattan. As any New Yorker knows, you can't walk from 2nd Avenue and 2nd Street to 6th Avenue and 6th Street by walking straight through buildings. The real distance walked is four blocks up and four blocks over. Mathematically, the Manhattan distance between two *n*-dimensional vectors $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_n)$ is

$$d = |a_1 - b_1| + |a_2 - b_2| + \ldots + |a_n - b_n|$$

The Mahout class that implements this measure is `ManhattanDistanceMeasure`.

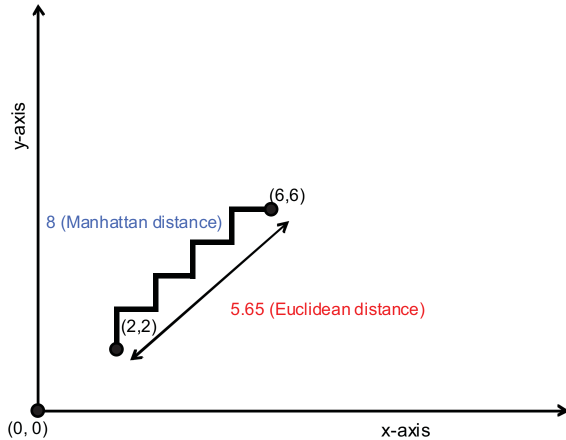


Figure 7.6 Difference between Euclidean and Manhattan distance measures. The Euclidean distance measure gives 5.65 as the distance between (2, 2) and (6, 6) whereas the Manhattan distance is 8.0.

7.4.4 Cosine distance measure

The cosine distance measure requires us to again think of points as vectors from the origin to those points. These vectors form an angle, θ, between them, as illustrated in figure 7.7.

When this angle is small, the vectors must be pointing in somewhat the same direction, and so in some sense the points are close. The cosine of this angle is near 1 when the angle is small, and decreases as it gets larger. The cosine distance equation subtracts the cosine value from 1 in order to give a proper distance, which is 0 when close and larger otherwise.

The formula for the cosine distance between *n*-dimensional vectors $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_n)$ is

$$d = 1 - \frac{(a_1 b_1 + a_2 b_2 + \ldots + a_n b_n)}{(\sqrt{(a_1^2 + a_2^2 + \ldots + a_n^2)}\sqrt{(b_1^2 + b_2^2 + \ldots + b_n^2)})}$$

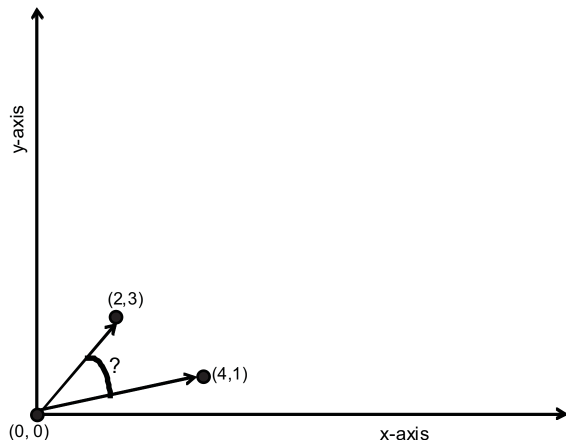


Figure 7.7 Cosine angle between the vectors (2, 3) and (4, 1), as calculated from the origin

Note that this measure of distance doesn't account for the length of the two vectors; all that matters is that the points are in the same direction from the origin. Also note that the cosine distance measure ranges from 0.0 (two vectors along the same direction) to 2.0 (two vectors in opposite directions).

The Mahout class that implements this measure is `CosineDistanceMeasure`.

7.4.5 Tanimoto distance measure

The cosine distance measure disregards the lengths of the vectors. This may work well for some data sets, but it'll lead to poor clustering in others where the relative lengths of the vectors contain valuable information.

For example, consider three vectors A (1.0, 1.0), B (3.0, 3.0), and C (3.5, 3.5). Because they point in the same direction, the cosine distance is 0.0 for any two of these vectors. Cosine distance doesn't capture the fact that B and C are in a sense closer. The Euclidean distance measure would reflect this, but it doesn't take account of the angle between the vectors—the fact that they're in the same direction. You might want, at times, a distance measure that reflects both.

The Tanimoto distance measure, also known as Jaccard's distance measure, captures the information about the angle and the relative distance between the points. The formula for the Tanimoto distance between two *n*-dimensional vectors ($a_1, a_2, \ldots, a_n$) and ($b_1, b_2, \ldots, b_n$) is

$$d = 1 - \frac{(a_1b_1 + a_2b_2 + \ldots + a_nb_n)}{\sqrt{(a_1^2 + a_2^2 + \ldots + a_n^2)} + \sqrt{(b_1^2 + b_2^2 + \ldots + b_n^2)} - (a_1b_1 + a_2b_2 + \ldots + a_nb_n)}$$

The Mahout class that implements this measure is `TanimotoDistanceMeasure`.

7.4.6 Weighted distance measure

Mahout also provides a `WeightedDistanceMeasure` class, and implementations of Euclidean and Manhattan distance measures that use it. A weighted distance measure is an advanced feature in Mahout that allows you to give weights to different dimensions in order to either increase or decrease the effect of a dimension on the value of the distance measure. The weights in a `WeightedDistanceMeasure` need to be serialized to a file in a `Vector` format.

For example, when calculating the distance between points in the x-y plane, suppose you wished to make the x coordinate twice as significant as the y coordinate. You could do so by doubling all x values, conceptually. To do this with a weighted distance measure, you construct a weight `Vector` with value of `2.0` in the zeroth index (for x) and `1.0` the first index (for y). This will affect specific distance measures differently but will in general make the distance value more sensitive to differences in x value.

7.5 Hello World again! Trying out various distance measures

Run the Hello World k-means clustering example using Euclidean, Manhattan, cosine, and Tanimoto distance measures, with k=2 (producing two clusters). The results of various runs are tabulated in table 7.1.

Table 7.1 Result of clustering using various distance measures

Distance measure	Number of iterations	Vectors[a] in cluster 0	Vectors in cluster 1
`EuclideanDistanceMeasure`	3	0, 1, 2, 3, 4	5, 6, 7, 8
`SquaredEuclideanDistanceMeasure`	5	0, 1, 2, 3, 4	5, 6, 7, 8
`ManhattanDistanceMeasure`	3	0, 1, 2, 3, 4	5, 6, 7, 8
`CosineDistanceMeasure`	1	1	0, 2, 3, 4, 5, 6, 7, 8
`TanimotoDistanceMeasure`	3	0, 1, 2, 3, 4	5, 6, 7, 8

a. These are the names/IDs of the vectors as given in the clustering Hello World source code.

The cosine distance measure clustering appears puzzling. In figure 7.2, you can see that only the point (2, 1) was at an angle greater than 45° from the x-axis. The clustering algorithm chose to put all other points, at 45° and below, in one cluster. This doesn't mean that the cosine distance measure is bad—only that it doesn't work well on this data set. In domains such as text clustering, for instance, it can work well.

`SquaredEuclideanDistanceMeasure` increased the number of iterations. This is because absolute distance values became larger when using that measure, and the algorithm was run using the same small value for the `convergenceThreshold`. As a result, it took a couple more iterations for the convergence to occur.

In future chapters, we look at more clustering methods, show how each of them is suited for various kinds of data, and explain how to optimize them, using various distance measures for both speed and quality.

7.6 Summary

In this chapter, we introduced the idea of clustering. We used an intuitive approach to clustering books in a library, and then we formalized notions of clustering using points in two dimensions. We created a small set of points in a plane and ran a simple k-means clustering example using `EuclideanDistanceMeasure`.

We then discussed the various distance measures found in Mahout. Armed with these, we reran the initial example and compared the clusters generated using each of the distance measures.

Before studying clustering algorithms in detail, we need to spend some time with another foundational concept in Mahout— how to represent data. That's up next.

8 Representing data

This chapter covers

- Representing data as a `Vector`
- Converting text documents into `Vector` form
- Normalizing data representations

To get good clustering, you need to understand the techniques of vectorization: the process of representing objects as `Vector`s. A `Vector` is a very simplified representation of data that can help clustering algorithms understand the object and help compute its similarity with other objects. This chapter explores various ways of converting different kinds of objects into `Vector`s.

In the last chapter, you got a taste of clustering. Books were clustered together based on the similarity of their words, and points in a two-dimensional plane were clustered together based on the distances between them. In reality, clustering could be applied to any kind of object, provided you could distinguish similar and dissimilar items. Images could be clustered based on their colors, the shapes in the images, or both. You could cluster photographs to perhaps try to distinguish photos of animals from those of humans. You could even cluster species of animals by their average sizes, weights, and number of legs to discover groupings automatically.

As humans, we can cluster these objects because we understand them, and we "just know" what is similar and what isn't. Computers, unfortunately, have no such intuition, so the clustering of anything by algorithms starts with representing the objects in a way that can be read by computers.

It turns out that it's quite practical, and flexible, to think of objects in terms of their measurable features or attributes. For example, we identified size and weight as salient features that could help produce some notion of animal similarity. Each object (animal) has a numeric value for these features.

We want to describe objects as sets of values, each associated to one of a set of distinct features, or dimensions—does this sound familiar? We've all but described a *vector* again. Although you're accustomed to thinking of vectors as arrows or points in space, they're just ordered lists of values. As such, they can easily represent objects.

We've already talked about how to cluster vectors in the previous chapter. But how can we represent the vectors in Mahout? How can we get from objects to vectors in the first place? That's what this chapter is all about. We continue with our discussion of how vectors derive from concepts in mathematics. We explain how the data that needs to be clustered is transformed into a vector form and encapsulated for Mahout to understand. We also go into text data in depth, and we discuss some important concepts such as weighting and normalization. Finally, we use all these concepts and vectorize Reuter's news dataset using the Mahout library.

8.1 Visualizing vectors

You might have encountered the word *vector* in many contexts. In physics, a vector denotes the direction and magnitude of a force, or the velocity of a moving object like a car. In mathematics, a vector is simply a point in space. Both these representations are conceptually very similar.

In two dimensions, vectors are represented as an ordered list of values, one for each dimension, like (4, 3). Both representations are illustrated in figure 8.1. We often name the first dimension x and the second y when dealing with two dimensions, but this won't matter for our purposes in Mahout. As far as we're concerned, a vector can have 2, 3, or 10,000 dimensions. The first is dimension 0, the next is dimension 1, and so on.

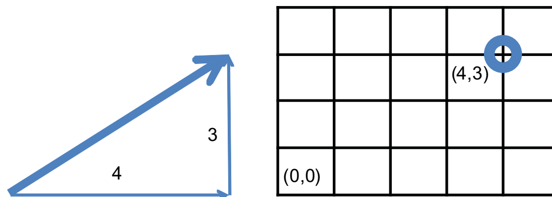


Figure 8.1 In physics, a vector can be thought of as a ray with a start point, direction, and length, and it represents quantities like velocity and acceleration. In geometry or space, a vector is just a point denoted by weights along each dimension. The direction and magnitude of the vector is by default assumed to be a ray from the origin (0,0).

8.1.1 Transforming data into vectors

In Mahout, vectors are implemented as three different classes, each of which is optimized for different scenarios: `DenseVector`, `RandomAccessSparseVector`, and `SequentialAccessSparseVector`.

- `DenseVector` can be thought of as an array of doubles, whose size is the number of features in the data. Because all the entries in the array are preallocated regardless of whether the value is 0 or not, we call it *dense*.
- `RandomAccessSparseVector` is implemented as a `HashMap` between an `integer` and a `double`, where only nonzero valued features are allocated. Hence, they're called as `SparseVectors`.
- `SequentialAccessSparseVector` is implemented as two parallel arrays, one of integers and the other of doubles. Only nonzero valued entries are kept in it. Unlike the `RandomAccessSparseVector`, which is optimized for random access, this one is optimized for linear reading.

Appendix B explains the differences between these clearly.

These three implementations provide you the flexibility to choose a vector class whose performance characteristics suit the nature of the data, the algorithm, and way data is accessed by it. The choice of implementation depends on the algorithm. If the algorithm does a lot of random insertions and updates of a vector's values, an implementation with fast random access like `DenseVector` or `RandomAccessSparseVector` is appropriate. On the other hand, for an algorithm like k-means clustering, which calculates the magnitudes of the vectors repeatedly, the `SequentialAccessSparseVector` implementation performs faster than the `RandomAccessSparseVector`.

To cluster objects, those objects first must be converted into vectors (they must be *vectorized*). The vectorization process is unique to each type of data, but because we're dealing with clustering in this part of the book, we talk about this data transformation with clustering in mind. We hope that by now the representation of an object as an *n*-dimensional vector of some kind is easy to accept. Objects must first be construed as a vector having as many dimensions as it has features. Let's look at this in the context of an example.

Suppose you want to cluster a bunch of apples. They're of different shapes, different sizes, and different colors (red, yellow, and green) as shown in figure 8.2. You first need to define a distance measure, which says that two apples are similar if they differ in few features, and by a small amount. So a small, round, red apple is more similar to a small, round, green one than a large, ovoid green one.

The process of vectorization starts with assigning features to dimensions. Let's say weight is feature (dimension) 0, color is 1, and size is 2. The vector of a small, round, red apple would therefore look like this:

```
[0 => 100 gram, 1 => red, 2 => small]
```

But this vector doesn't have all the numeric values yet, and it needs to.

No. 8

For dimension 0, you need to express weight as a number. This could simply be the measured weight in grams or kilograms. Size, dimension 2, doesn't necessarily mean the same as weight. For all we know, the green apple could be denser than the red apple due to its freshness. Density could also be used, provided we have the instruments to measure it. Size, on the other hand, could be specified as user-perceived numbers: small apples could be of size value 1, medium apples could be size 2, and large ones size 3.

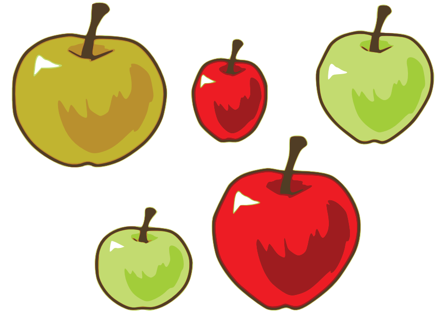


Figure 8.2 Apples of different sizes and colors needs to be converted into an appropriate vector form. The trick is to figure out how the different features of the apples translate into a decimal value.

What about color, dimension 1? You could arbitrarily assign numbers to the colors, like red = 0.0, green = 1.0, yellow = 2.0. This is a crude representation; it'll work in many cases, but it fails to reflect the fact that yellow is a color between red and green in the visible spectrum. We could fix that by changing mappings, but perhaps it'd be better to use something like the wavelength of the color (400–650 nm). This maps color to a meaningful and objective dimension value.

Using these measures as properties of the apples, the vectors for some apples are described in table 8.1.

Table 8.1 set of apples of different weight, sizes, and colors converted to vectors

Apple	Weight (kg) (0)	Color (1)	Size (2)	Vector
Small, round, green	0.11	510	1	[0.11, 510, 1]
Large, oval, red	0.23	650	3	[0.23, 650, 3]
Small, elongated, red	0.09	630	1	[0.09, 630, 1]
Large, round, yellow	0.25	590	3	[0.25, 590, 3]
Medium, oval, green	0.18	520	2	[0.18, 520, 2]

If you weren't interested in clustering apples based on similarity in color, you could've kept each color in a different dimension. That is, red would be dimension 1, green dimension 3, and yellow dimension 4. If the apple is red, red would have value 1 and the others 0. You could then store these vectors in a sparse format, and the distance measure would consider only the presence of nonzero values in these dimensions and cluster together those apples that are of the same color.

One possible problem with our chosen mappings to dimension values is that the values in dimension 1 are much larger than the others. If we applied a simple distance-based metric to determine similarity between these vectors, color differences

would dominate the results. A relatively small color difference of 10 nm is treated as equal to a huge size difference of 10. Weighting the different dimensions solves this problem.

The importance of weighting is discussed in section 8.2 where you'll generate vectors from text documents. Not all words in a document represent the document to the same extent. The weighting technique helps magnify the more important words and deemphasize the least important ones.

8.1.2 Preparing vectors for use by Mahout

Having established how to encode apples as vectors, let's look at how to prepare vectors specifically for consumption by Mahout algorithms. An implementation of `Vector` is instantiated and filled in for each object; then, all `Vectors` are written to a file in the `SequenceFile` format, which is read by the Mahout algorithms. `SequenceFile` is a format from the Hadoop library that encodes a series of key-value pairs. Keys must implement `WritableComparable` from Hadoop, and values must implement `Writable`. These are Hadoop's equivalent of Java's `Comparable` and `Serializable` interfaces.

For an example, let's use the vector's name or description as a key, and the vector itself as the value. Mahout's `Vector` classes don't implement the `Writable` interface to avoid coupling them directly to Hadoop, but the `VectorWritable` wrapper class can be used to wrap a `Vector` and make it `Writable`. The Mahout `Vector` can be written to the `SequenceFile` using the `VectorWritable` class, as follows.

Listing 8.1 Creating vectors of various apples

```
public static void main(String args[]) throws Exception {
  List<NamedVector> apples = new ArrayList<NamedVector>();

  NamedVector apple;
  apple = new NamedVector(
      new DenseVector(new double[] {0.11, 510, 1}),
      "Small round green apple");                      <- Associates a name with the vector
  apples.add(apple);
  apple = new NamedVector(
    new DenseVector(new double[] {0.23, 650, 3}),
      "Large oval red apple");
  apples.add(apple);
  apple = new NamedVector(
    new DenseVector(new double[] {0.09, 630, 1}),
      "Small elongated red apple");
  apples.add(apple);
  apple = new NamedVector(
    new DenseVector(new double[] {0.25, 590, 3}),
      "Large round yellow apple");
  apples.add(apple);
  apple = new NamedVector(
    new DenseVector(new double[] {0.18, 520, 2}),
      "Medium oval green apple");

  Configuration conf = new Configuration();
  FileSystem fs = FileSystem.get(conf);
```

```
  Path path = new Path("appledata/apples");
  SequenceFile.Writer writer = new SequenceFile.Writer(fs, conf,
      path, Text.class, VectorWritable.class);
  VectorWritable vec = new VectorWritable();
  for (NamedVector vector : apples) {
    vec.set(vector);
    writer.append(new Text(vector.getName()), vec);      <- Serializes vector data
  }
  writer.close();

  SequenceFile.Reader reader = new SequenceFile.Reader(fs,
      new Path("appledata/apples"), conf);

  Text key = new Text();
  VectorWritable value = new VectorWritable();
  while (reader.next(key, value)) {                       <- Deserializes vector data

    System.out.println(key.toString() + " "
        + value.get().asFormatString());
  }
  reader.close();
}
```

The process of selecting the features of an object and mapping them to numbers is known as *feature selection*. The process of encoding features as a vector is *vectorization*.

Any kind of object can be converted to vector form using reasonable approximations of the feature values, as we did for apples. Now let's turn to vectorizing one particularly interesting type of object: text documents.

8.2 Representing text documents as vectors

Text content in the digital form is exploding. The Google search engine alone indexes over 20 billion web documents. That's just a fraction of the publicly crawlable information. The estimated size of text data (both public and private) could go well beyond the petabyte range: that's a 1 followed by 15 zeros. There's a huge opportunity here for machine learning algorithms like clustering and classification to figure out some structure and meaning in this unstructured world, and learning the art of text vectorization is the first step into it.

The vector space model (VSM) is the common way of vectorizing text documents. First, imagine the set of all words that could be encountered in a series of documents being vectorized. This set might be all words that appear at least once in any of the documents. Imagine each word being assigned a number, which is the dimension it'll occupy in document vectors.

For example, if the word *horse* is assigned to the 39,905th index of the vector, the word *horse* will correspond to the 39,905th dimension of document vectors. A document's vectorized form merely consists, then, of the number of times each word occurs in the document, and that value is stored in the vector along that word's dimension. The dimension of these document vectors can be very large. The maximum number of dimensions possible is the *cardinality* of the vector. Because the count of all possible words or tokens is unimaginably large, text vectors are usually assumed to have infinite dimensions.

The value of the vector dimension for a word is usually the number of occurrences of the word in the document. This is known as term frequency (TF) weighting. Note that values in a vector are also referred to as weights in this field; you may see references to weighting instead of values. The number of unique words that appear in one document is typically small compared to the number of unique words that appear in *any* document in a collection being processed. As a result, these high-dimension document vectors are quite sparse.

In clustering, we frequently try to find the similarity between two documents based on a distance measure. In typical English-language documents, the most frequent words will be *a*, *an*, *the*, *who*, *what*, *are*, *is*, *was*, and so on. Such words are called stopwords. If you calculate the distance between two document vectors using any distance measure, you'll see that the distance value is dominated by the weights of these frequent words.

This is the same problem noted before with apples and color. This effect is undesirable because it implies that two documents are similar mostly because words like *a*, *an*, and *the* occur in both. But, intuitively, we think of two documents as similar if they talk about similar topics, and words that signal a topic are usually the rarer words like *enzyme*, *legislation*, *Jordan* and so on. This makes simple term-frequency based weighting undesirable for clustering and for applications where document similarity is to be calculated.

Fortunately, weighting can be modified with a very simple but effect trick to fix these shortcomings, as we discuss next.

8.2.1 *Improving weighting with TF-IDF*

Term frequency–inverse document frequency (TF-IDF) weighting is a widely used improvement on simple term-frequency weighting. The IDF part is the improvement; instead of simply using term frequency as the value in the vector, this value is multiplied by the inverse of the term's document frequency. That is, its value is *reduced* more for words used frequently across all the documents in the dataset than for infrequently used words.

To illustrate this, say that a document has words w_1, w_2, ..., w_n with frequencies f_1, f_2, …, f_n. The term frequency (TF_i) of word w_i is the frequency f_i.

To calculate the inverse document frequency, the document frequency (DF) for each word is first calculated. Document frequency is the number of documents the word occurs in. The number of times a word occurs in a document isn't counted in document frequency. Then, the inverse document frequency or IDF_i for a word, w_i, is

$$IDF_i = \frac{1}{DF_i}$$

If a word occurs frequently in a collection of documents, its DF value is large and its IDF value is small; so small that it reduces the weight when multiplied. In such cases, it's best to normalize the IDF score by multiplying it by a constant number. Usually it's multiplied by the document count (N), so the IDF equation will look like this:

$$IDF_i = \frac{N}{DF_i}$$

Therefore, the weight W_i of a word w_i in a document vector is

$$W_i = TF_i \cdot IDF_i = TF_i \cdot \frac{N}{DF_i}$$

The IDF value in the preceding form is still not ideal, because it masks the effect of TF on the final term weight. To reduce this problem, a usual practice is to use the logarithm of the IDF value instead:

$$IDF_i = \log\frac{N}{DF_i}$$

Thus, the TF-IDF weight W_i for a word, w_i, becomes

$$W_i = TF_i \cdot \log\frac{N}{DF_i}$$

That is, the document vector will have this value at the dimension for word *i*. This is the classic TF-IDF weighting. Stop words get a small weight, and terms that occur infrequently get a large weight. The important words, or the topic words, usually have a high TF and a somewhat large IDF, so the product of the two becomes a larger value, thereby giving more importance to these words in the vector produced.

The basic assumption of the vector space model (VSM) is that the words are dimensions and therefore are orthogonal to each other. In other words, VSM assumes that the occurrences of words are independent of each other, in the same sense that a point's *x* coordinate is entirely independent of its *y* coordinate, in two dimensions. By intuition you know that this assumption is wrong in many cases. For example, the word *Cola* has higher probability of occurring along with the word *Coca*, so these words aren't completely independent. Other models try to consider word dependencies. One well-known technique is latent semantic indexing (LSI), which detects dimensions that seem to go together and merges them into a single one. Due to the reduction in dimensions, the clustering computations are faster. It also improves the quality of clustering, because there is now a single good feature for the document object that dominates grouping really well.

At the time of writing, Mahout doesn't yet implement LSI, but TF-IDF has proved to work remarkably well even with the independence assumption. Mahout currently provides a solution to the problem of word dependencies by using a method called collocation or *n*-gram generation, which we look at next.

8.2.2 *Accounting for word dependencies with* n-*gram collocations*

A group of words in a sequence is called an *n*-gram. A single word can be called a unigram, and two words like *Coca Cola* can be considered a single unit and called a bigram. Combinations of three and more terms can be called trigrams, 4-grams, 5-grams, and so on. Classic TF-IDF weighting assumes that words occur independently of

other words, but vectors created using this method usually lack the ability to identify key features of documents, which may be dependent.

To circumvent this problem, Mahout implements techniques to identify groups of words that have an unusually high probability of occurring together, such as *Martin Luther King Jr* or *Coca Cola*. Instead of creating vectors where dimensions map to single words (unigrams), you could as easily create vectors where dimensions map to bigrams—or even both. TF-IDF can then work its magic as before.

From a sentence consisting of multiple words, you can generate all *n*-grams by selecting sequential blocks of *n* words. This exercise will generate many *n*-grams, most of which don't represent meaningful units. For example, from the sentence, "It was the best of times, it was the worst of times," we can generate the following bigrams:

It was
was the
the best
best of
of times
times it
it was
was the
the worst
worst of
of times

Some of these are good combinations for generating document vectors ("the best," "the worst"), but some of them aren't ("was the"). If you combine the unigrams and bigrams from a document and generate weights using TF-IDF, you'll end up with many meaningless bigrams having large weights because of their large IDF. This is quite undesirable.

Mahout solves this problem by passing the *n*-grams through something called a log-likelihood test, which can determine whether two words occurred together by chance or because they form a significant unit. It selects the most significant ones and prunes away the least significant ones. Using the remaining *n*-grams, the TF-IDF weighting scheme is applied and vectors are produced. In this way, significant bigrams like *Coca Cola* can be more properly accounted for in a TF-IDF weighting.

In Mahout, text documents are converted to vectors using TF-IDF weighting and *n*-gram collocation using the `DictionaryVectorizer` class. In the next section you'll see how to start from a directory full of documents and create TF-IDF weighted vectors.

8.3 Generating vectors from documents

We now examine two important tools that generate vectors from text documents. The first is the `SequenceFilesFromDirectory` class, which generates an intermediate document representation in `SequenceFile` format from text documents under a directory structure. The second, `SparseVectorsFromSequenceFiles`, converts the text documents in `SequenceFile` format to vectors using either TF or TF-IDF weighting

with *n*-gram generation. The intermediate `SequenceFile` is keyed by document ID; the value is the document's text content. Starting from a directory of text documents, with each file containing a full document, you'll use these classes to convert the documents to vectors.

For this example, we use the Reuters-21578 news collection.[1] It's a widely used dataset for machine learning research. The data was originally collected and labeled by Carnegie Group and Reuters in the course of developing the CONSTRUE text categorization system. The Reuters-21578 collection is distributed in 22 files, each of which contains 1,000 documents, except for the last (reut2-021.sgm), which contains 578 documents.

The files are in SGML format, which is similar to XML. We could create a parser for the SGML files, write the document ID and document text into `SequenceFiles`, and use `SparseVectorsFromSequenceFiles` to convert them to vectors. But a much quicker way is to reuse the Reuters parser given in the Lucene benchmark JAR file. Because its bundled along with Mahout, all you need to do is change to the examples/ directory under the Mahout source tree and run the `org.apache.lucene.benchmark.utils.ExtractReuters` class.

Before doing this, download the Reuters collection from the website and extract it in the reuters/ folder under examples/. Run the Reuters extraction code from the examples directory as follows:

```
mvn -e -q exec:java
-Dexec.mainClass="org.apache.lucene.benchmark.utils.ExtractReuters"
-Dexec.args="reuters/ reuters-extracted/"
```

Using the extracted folder, run the `SequenceFileFromDirectory` class. You can use the launcher script from the Mahout root directory to do the same:

```
bin/mahout seqdirectory -c UTF-8
-i examples/reuters-extracted/ -o reuters-seqfiles
```

> **TIP** You may need to set the `JAVA_HOME` variable in your environment to run the launcher script.

This will write the Reuters articles in the `SequenceFile` format. Now the only step left is to convert this data to vectors. To do that, run the `SparseVectorsFromSequenceFiles` class using the Mahout launcher script:

```
bin/mahout seq2sparse -i reuters-seqfiles/ -o reuters-vectors -ow
```

> **TIP** In Mahout, the `-ow` flag is used to denote whether or not to overwrite the output folder. Because Mahout deals with huge datasets, it takes time to generate the output for each algorithm. This flag will prevent accidental deletion of any output that took hours to produce.

[1] The Reuters-21578 test collection can be found at http://www.daviddlewis.com/resources/testcollections/reuters21578/. The direct link for downloading the data is http://www.daviddlewis.com/resources/testcollections/reuters21578/reuters21578.tar.gz.

The `seq2sparse` command in the Mahout launcher script reads the Reuters data from `SequenceFile` and writes the vector generated by the dictionary-based vectorizer to the output folder using the default options listed shortly in table 8.2.

Table 8.2 Important flags for the Mahout dictionary-based vectorizer and their default values

Option	Flag	Description	Default value
Overwrite (`bool`)	`-ow`	If set, the output folder is overwritten. If not set, the output folder is created if the folder doesn't exist. If the output folder does exist, the job fails and an error is thrown. Default is unset.	N/A
Lucene analyzer name (`String`)	`-a`	The class name of the analyzer to use.	`org.apache.lucene .analysis.standard .StandardAnalyzer`
Chunk size (`int`)	`-chunk`	The chunk size in MB. For large document collections (sizes in GBs and TBs), you won't be able to load the entire dictionary into memory during vectorization, so you can split the dictionary into chunks of the specified size and perform the vectorization in multiple stages. It's recommended you keep this size to 80 percent of the Java heap size of the Hadoop child nodes to prevent the vectorizer from hitting the heap limit.	`100`
Weighting (`String`)	`-wt`	The weighting scheme to use: `tf` for term-frequency based weighting and `tfidf` for TF-IDF based weighting.	`tfidf`
Minimum support (`int`)	`-s`	The minimum frequency of the term in the entire collection to be considered as a part of the dictionary file. Terms with lesser frequency are ignored.	`2`
Minimum document frequency (`int`)	`-md`	The minimum number of documents the term should occur in to be considered a part of the dictionary file. Any term with lesser frequency is ignored.	`1`
Max document frequency percentage (`int`)	`-x`	The maximum number of documents the term should occur in to be considered a part of the dictionary file. This is a mechanism to prune out high frequency terms (stop-words). Any word that occurs in more than the specified percentage of documents is ignored.	`99`
N-gram size (`int`)	`-ng`	The maximum size of *n*-grams to be selected from the collection of documents.	`1`

Table 8.2 Important flags for the Mahout dictionary-based vectorizer and their default values *(continued)*

Option	Flag	Description	Default value
Minimum log-likelihood ratio (LLR) (`float`)	`-ml`	This flag works only when *n*-gram size is greater than 1. Very significant *n*-grams have large scores, such as 1000; less significant ones have lower scores. Although there's no specific method for choosing this value, the rule of thumb is that *n*-grams with a LLR value less than 1.0 are irrelevant.	`1.0`
Normalization (`float`)	`-n`	The normalization value to use in the L_p space. A detailed explanation of normalization is given in section 8.4. The default scheme is to not normalize the weights.	`0`
Number of reducers (`int`)	`-nr`	The number of reducer tasks to execute in parallel. This flag is useful when running a dictionary vectorizer on a Hadoop cluster. Setting this to the maximum number of nodes in the cluster gives maximum performance. Setting this value higher than the number of cluster nodes leads to a slight decrease in performance. For more details, read the Hadoop documentation on setting the optimum number of reducers.	`1`
Create sequential access sparse vectors (`bool`)	`-seq`	If set, the output vectors are created as `SequentialAccessSparseVector`s. By default the dictionary vectorizer generates `RandomAccessSparseVector`s. The former gives higher performance on certain algorithms like k-means and SVD due to the sequential nature of vector operations. By default the flag is unset.	N/A

Inspect the folder that's produced using these command-line commands:

```
$ ls reuters-vectors/
df-count/
dictionary.file-0
frequency.file-0
tfidf-vectors/
tf-vectors/
tokenized-documents/

wordcount/
```

In the output folder, you'll find a dictionary file and several directories. The dictionary file contains the mapping between a term and its integer ID. This file is useful when reading the output of different algorithms, so you need to retain it. The other folders are intermediate folders generated during the vectorization process, which happens in multiple steps (multiple MapReduce jobs).

In the first step, the text documents are tokenized—they're split into individual words using the Lucene StandardAnalyzer and stored in the tokenized-documents/ folder. The word-counting step—the *n*-gram generation step (which in this case only counts unigrams)—iterates through the tokenized documents and generates a set of important words from the collection. The third step converts the tokenized documents into vectors using the term-frequency weight, thus creating TF vectors. By default, the vectorizer uses the TF-IDF weighting, so two more steps happen after this: the document-frequency (DF) counting job, and the TF-IDF vector creation.

The TF-IDF weighted vectorized documents are found in the tfidf-vectors/ folder. For most applications, you need just this folder and the dictionary file.

Let's revisit the Reuters SequenceFiles and generate a vector dataset using non-default values. You'll use the following non-default flag values:

- -a—Use org.apache.lucene.analysis.WhitespaceAnalyzer to tokenize words based on the whitespace characters between them.
- -chunk—Use a chunk size of 200 MB. This value won't produce any effect on the Reuters data, because the dictionary sizes are usually in the 1 MB range.
- -wt—Use the tfidf weighting method.
- -s—Use a minimum support value of 5.
- -md—Use a minimum document frequency value of 3.
- -x—Use a maximum document frequency percentage of 90 percent to aggressively prune away high-frequency words.
- -ng—Use an *n*-gram size of 2 to generate both unigrams and bigrams.
- -ml—Use a minimum log-likelihood ratio (LLR) value of 50 to keep only very significant bigrams.
- -seq—Set the SequentialAccessSparseVectors flag.

Leave the normalization flag (-n) unset. You'll get back to this flag in the next section.

Run the vectorizer using the preceding options in the Mahout launcher script:

```
bin/mahout seq2sparse -i reuters-seqfiles/ -o reuters-vectors-bigram -ow
-a org.apache.lucene.analysis.WhitespaceAnalyzer
-chunk 200 -wt tfidf -s 5 -md 3 -x 90 -ng 2 -ml 50 -seq
```

The dictionary file sizes from this vectorization job have increased from 654 KB to 1.2 MB. Though we pruned away more unigrams based on frequency, we added almost double the number of bigrams even after filtering using the LLR threshold value. The dictionary size goes up to 2 MB upon including trigrams. At least it's only grown linearly as we move from bigrams to trigrams and onwards; this is attributable to the LLR-based filtering process. Without this, the dictionary size would've grown exponentially.

At this point, you're almost ready to try any clustering algorithm Mahout has to offer. There is just one more concept in text vectorization that's important to understand: normalization. We explore it next.

8.4 Improving quality of vectors using normalization

Normalization, here, is a process of cleaning up edge cases—data with unusual characteristics that skew results disproportionately. For example, when calculating the similarity between documents based on some distance measure, it isn't uncommon that a few documents pop up as if they're similar to all the other documents in the collection, but on closer inspection, you'll find that this happened because the document is large, and its vector has many nonzero dimensions, causing it to be close to many smaller documents. Somehow, we need to negate the effect of varying sizes of the vectors when calculating similarity. This process of decreasing the magnitude of large vectors and increasing the magnitude of smaller vectors is called *normalization.*

In Mahout, normalization uses what is known in statistics as a *p*-norm. For example, the *p*-norm of a 3-dimensional vector, [x, y, z], is

$$\frac{x}{(|x|^p + |y|^p + |z|^p)^{1/p}}, \frac{y}{(|x|^p + |y|^p + |z|^p)^{1/p}}, \frac{z}{(|x|^p + |y|^p + |z|^p)^{1/p}}$$

The expression $(|x|^p + |y|^p + |z|^p)^{1/p}$ is known as the norm of a vector; here, we've merely divided each dimension's value by this number. The parameter p could be any value greater than 0. The 1-norm, or Manhattan norm, of a vector is the vector divided by the sum of the weights of all the dimensions:

$$\frac{x}{|x| + |y| + |z|}, \frac{y}{|x| + |y| + |z|}, \frac{z}{|x| + |y| + |z|}$$

The 2-norm, or Euclidean norm, is the vector divided by the magnitude of the vector—this magnitude is the *length* of the vector as we're accustomed to understanding it:

$$\frac{x}{\sqrt{x^2 + y^2 + z^2}}, \frac{y}{\sqrt{x^2 + y^2 + z^2}}, \frac{z}{\sqrt{x^2 + y^2 + z^2}}$$

The infinite norm is simply the vector divided by the weight of the largest magnitude dimension:

$$\frac{x}{\max(|x|, |y|, |z|)}, \frac{y}{\max(|x|, |y|, |z|)}, \frac{z}{\max(|x|, |y|, |z|)}$$

The norm power (p) you choose will depend upon the type of operations done on the vector. If the distance measure used is the Manhattan distance measure, the 1-norm will often yield better results with the data. Similarly, if the cosine of the Euclidean distance measure is being used to calculate similarity, the 2-norm version of the vectors yields better results. For best results, the normalization ought to relate to the notion of distance used in the similarity metric.

Note that the p in p-norm can be any rational number, so 3/4, 5/3, and 7/5 are all valid powers of normalization. In the dictionary vectorizer, the power is set using the `-norm` flag. A value of `INF` means an infinite norm. Generating the 2-normalized bigram vectors is as easy as running the Mahout launcher using the `seq2sparse` command with the `-n` flag set to `2`:

```
bin/mahout seq2sparse -i reuters-seqfiles/ -o reuters-normalized-bigram -ow
-a org.apache.lucene.analysis.WhitespaceAnalyzer
-chunk 200 -wt tfidf -s 5 -md 3 -x 90 -ng 2 -ml 50 -seq -n 2
```

Normalization improves the quality of clustering a little. Further refinement in the quality of clustering is achieved by using problem-specific distance measures and appropriate algorithms. In the next chapter, we take you on an elephant-ride past the various clustering algorithms in Mahout.

8.5 Summary

In this chapter, you learned about the most important data representation scheme used by machine learning algorithms like clustering: the `Vector` format. There are two types of `Vector` implementations in Mahout, sparse and dense vectors. Dense vectors are implemented by the `DenseVector` class; `RandomAccessSparseVector` is a sparse implementation designed for applications requiring fast random reads, and the `SequentialAccessSparseVector` is designed for applications requiring fast sequential reads. You can choose one implementation over the other based on your algorithm's access patterns.

You learned how to map important features of an object to numerical values and thereby create vectors representing different types of objects (apples in our example). The vectors were then written to and read from a `SequenceFile`, which is the format used by all the clustering algorithms in Mahout.

Text documents are frequently used in context of clustering. You saw how text documents could be represented as `Vectors` using the vector space model (VSM). The TF-IDF weighting scheme proved to be a simple and elegant way to remove the negative impact of stop-words during clustering. The assumption of the independence of words in the classic TF-IDF weighting scheme removes some important features from the text, but the collocation-based *n*-gram generation in Mahout solves this problem to a great extent by identifying significant groups of words using a log-likelihood ratio test. You saw how the Mahout dictionary-based vectorizer converted the Reuters news collections to vectors with ease.

Finally, you saw that the length of text documents negatively affects the quality of distance measures. The *p*-normalization method implemented in the dictionary vectorizer solves this problem by readjusting the weights of the vectors by dividing by the *p*-norm of the vector.

Using the Reuters vector dataset, we can do clustering with different techniques, each having its pros and cons. We explore these techniques in the next chapter.

9 Clustering algorithms in Mahout

This chapter covers

- K-means clustering
- Centroid generation using canopy clustering
- Fuzzy k-means clustering and Dirichlet clustering
- Topic modeling using latent Dirichlet allocation as a variant of clustering

Now that you know how input data is represented as `Vectors` and how `SequenceFiles` are created as input for the clustering algorithms, you're ready to explore the various clustering algorithms that Mahout provides. There are many clustering algorithms in Mahout, and some work well for a given data set whereas others don't. K-means is a generic clustering algorithm that can be molded easily to fit almost all situations. It's also simple to understand and can easily be executed on parallel computers.

Therefore, before going into the details of various clustering algorithms, it's best to get some hands-on experience using the k-means algorithm. Then it becomes easier to understand the shortcomings and pitfalls of other less generic techniques, and

see how they can achieve better clustering of data in particular situations. You'll use the k-means algorithm to cluster news articles and then improve the clustering quality using other techniques. You'll then learn how the value of *k* in k-means can be inferred using canopy clustering. With this knowledge, you'll create a clustering pipeline for a news aggregation website to get a better feel for real-world problems in clustering.

Once you're comfortable with k-means, you'll look at some of the shortcomings of k-means and see how other special types of clustering algorithms can be used to fill those gaps. Fuzzy k-means and Dirichlet clustering will be discussed in that context. Finally, you'll explore latent Dirichlet allocation (LDA)—an algorithm that closely resembles clustering but that achieves something far more interesting.

There's a lot to cover, so let's not waste any time. Let's jump right into the world of clustering with the k-means algorithm.

9.1 K-means clustering

K-means is to clustering as Vicks is to cough syrup. It's a simple algorithm and it's more than 50 years old. Stuart Lloyd first proposed the standard algorithm in 1957 as a technique for pulse code modulation, but it wasn't until 1982 that it was published.[1] It's widely used as a clustering algorithm in many fields of science. The algorithm requires the user to set the number of clusters, *k*, as the input parameter. You've already seen the algorithm in action in chapter 7 when you clustered a set of points in a 2-D plane. Let's look at the details to understand the algorithm.

The k-means algorithm puts a hard limitation on the number of clusters, *k*. This limitation might cause you to question the quality of this method, but fear not. This algorithm has proven to work very well for a wide range of real-world problems over the 29 years of its existence. Even if your estimate for the value *k* is suboptimal, the clustering quality isn't affected much by it.

Say you're clustering news articles to get top-level categories like politics, science, and sports. For that, you might want to choose a small value of *k*, perhaps in the range of 10 to 20. If fine-grained topics are needed, a larger value of *k*, like 50 to 100 is necessary. Suppose there are 1,000,000 news articles in your database and you're trying to find groups of articles about the same story. The number of such related stories would be much smaller than the entire corpus—maybe in the range of 100 articles per cluster. This means you would need a *k* value of 10,000 to generate such a distribution. That will surely test the scalability of clustering, and this is where Mahout shines.

For good quality clustering using k-means, you'll need to estimate a value for *k*. An approximate way of estimating *k* is to figure it out based on the data you have and the size of clusters you need. In the preceding example, where we have about a million news articles, if there are an average of 500 news articles published about every unique story, you should start your clustering with a *k* value of about 2,000 (1,000,000/500).

[1] Stuart P. Lloyd, "Least Squares Quantization in pcm", *IEEE Transactions on Information Theory*, IT-28, 2: 129–37. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.131.1338.

This is a crude way of estimating the number of clusters. Nevertheless, the k-means algorithm generates decent clustering even with this approximation. The type of distance measure used is the main determining factor in the quality of k-means clusters. In chapter 7, we mentioned the various kinds of distance measures in Mahout. It's worthwhile to review them and to test their effects on the examples in this chapter.

9.1.1 All you need to know about k-means

Let's look at the k-means algorithm in detail. Suppose we have n points that we need to cluster into k groups. The k-means algorithm will start with an initial set of k centroid points. The algorithm does multiple rounds of processing and refines the centroid locations until the iteration max-limit criterion is reached or until the centroids converge to a fixed point from which they don't move very much.

A single k-means iteration is illustrated in figure 9.1. The actual algorithm is a series of such iterations.

No. 9

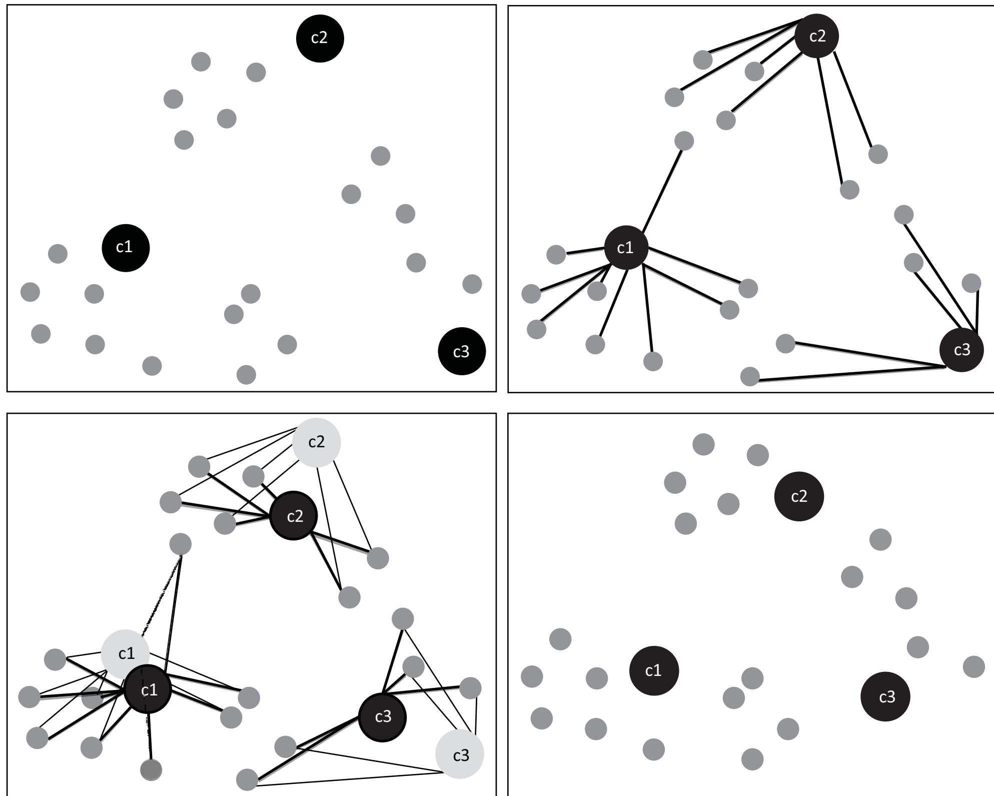


Figure 9.1 K-means clustering in action. Starting with three random points as centroids (top left), the map stage (top right) assigns each point to the cluster nearest to it. In the reduce stage (bottom left), the associated points are averaged out to produce the new location of the centroid, leaving you with the final configuration (bottom right). After each iteration, the final configuration is fed back into the same loop until the centroids come to rest at their final positions.

There are two steps in this algorithm. The first step finds the points that are nearest to each centroid point and assigns them to that specific cluster. The second step recalculates the centroid point using the average of the coordinates of all the points in that cluster.

This two-step algorithm is a classic example of an expectation maximization (EM) algorithm. In an EM algorithm, the two steps are processed repeatedly until convergence is reached. The first step, known as the expectation (E) step, finds the expected points associated with a cluster. The second step, known as the maximization (M) step, improves the estimation of the cluster center using the knowledge from the E step. A complete explanation of EM is beyond the scope of this book, but plenty of resources can be found online.[2]

Now that you understand the k-means technique, let's look at the important k-means classes in Mahout and run a simple clustering example.

9.1.2 Running k-means clustering

The k-means clustering algorithm is run using either the `KMeansClusterer` or the `KMeansDriver` class. The former does an in-memory clustering of the points, whereas the latter is an entry point to launch k-means as a MapReduce job. Both methods can be run like a regular Java program and can read and write data from the disk. They can also be executed on an Apache Hadoop cluster, reading data from and writing it to a distributed filesystem.

For this example, you're going to use a random point generator function to create the points. It generates the points in `Vector` format as a normal distribution around a given center. You'll cluster these points using the in-memory k-means clustering implementation in Mahout.

The `generateSamples` function in listing 9.1 takes a center, for example (1, 1), the standard deviation (2), and creates a set of n (400) random points around the center, which behaves like a normal distribution. You'll similarly create two other sets with centers (1, 0) and (0, 2) and standard deviations 0.5 and 0.1 respectively. Listing 9.1 runs `KMeansClusterer` using the following parameters:

- The input points are in `List<Vector>` format.
- The `DistanceMeasure` is `EuclideanDistanceMeasure`.
- The threshold of convergence is 0.01.
- The number of clusters, k, is 3.
- The clusters were chosen using `RandomPointsUtil`, as in the Hello World example in chapter 7.

[2] Frank Dellaert explains the EM algorithm in terms of lower bound maximization in "The Expectation Maximization Algorithm," http://www.cc.gatech.edu/~dellaert/em-paper.pdf. Wikipedia also has an article on the algorithm: http://en.wikipedia.org/wiki/Expectation-maximization_algorithm.

Listing 9.1 In-memory clustering example using the k-means algorithm

```
private static void generateSamples(List<Vector> vectors, int num,
  double mx, double my, double sd) {
     for (int i = 0; i < num; i++) {
       vectors.add(new DenseVector(
         new double[] {
           UncommonDistributions.rNorm(mx, sd),
           UncommonDistributions.rNorm(my, sd)
         }
       ));
     }
   }
public static void main(String[] args) {
  List<Vector> sampleData = new ArrayList<Vector>();

  generateSamples(sampleData, 400, 1, 1, 3);          Generate three
  generateSamples(sampleData, 300, 1, 0, 0.5);        sets of points
  generateSamples(sampleData, 300, 0, 2, 0.1);

  int k = 3;

  List<Vector> randomPoints = RandomPointsUtil.chooseRandomPoints(
    sampleData, k);
     List<Cluster> clusters = new ArrayList<Cluster>();

     int clusterId = 0;
     for (Vector v : randomPoints) {
       clusters.add(new Cluster(v, clusterId++,
          new EuclideanDistanceMeasure()));
     }

     List<List<Cluster>> finalClusters
         = KMeansClusterer.clusterPoints(sampleData, clusters,
             new EuclideanDistanceMeasure(), 3, 0.01);     Run
     for(Cluster cluster : finalClusters.get(                KMeansClusterer
         finalClusters.size() - 1)) {
       System.out.println("Cluster id: " + cluster.getId()
           + " center: " +                                 Read center of
           cluster.getCenter().asFormatString());          cluster and print it
     }
   }
 }
```

The `DisplayKMeans` class in Mahout's `mahout-examples` module is a great tool for visualizing the algorithm in a 2-dimensional plane. It shows how the clusters shift their positions after each iteration. It's also a great example of how clustering is done using `KMeansClusterer`. Run `DisplayKMeans` as a Java Swing application, and view the output of the example, as shown in figure 9.2.

Note that the k-means in-memory clustering implementation works with a list of `Vector` objects. The amount of memory used by this program depends on the total size of all the vectors. The sizes of clusters are larger than the size of the vectors in the case of sparse vectors, or they're the same size for dense vectors. As a rule of thumb, assume that the amount of memory needed includes the total size of all of the input

vectors plus the size of the *k* centers. If the data is huge, you can't run this implementation.

This is where MapReduce shines. Using the MapReduce infrastructure, you can split this clustering algorithm up to run on multiple machines, with each mapper getting a subset of the points. The mapper jobs will partially compute the nearest cluster by reading the input points in a streaming fashion.

The MapReduce version of the k-means algorithm is designed to run on a Hadoop cluster, but it runs quite efficiently without one. Mahout is compiled against Hadoop code, which means you could run the same implementation without a Hadoop cluster directly from within Java and mimic Hadoop's single machine behavior.

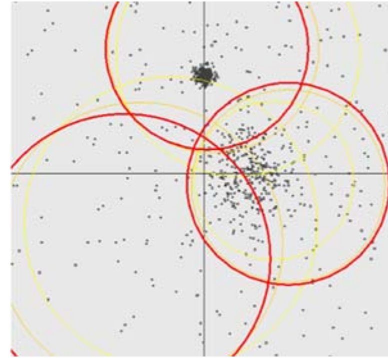


Figure 9.2 In this example of k-means clustering we start with *k* set to 3 and try to cluster three normal distributions. The thin lines denote the clusters estimated in previous iterations—you can clearly see the clusters shifting.

UNDERSTANDING THE K-MEANS CLUSTERING MAPREDUCE JOB

In Mahout, the MapReduce version of the k-means algorithm is instantiated using the `KMeansDriver` class. The class has just a single entry point—the `runJob` method.

You've already seen k-means in action in chapter 7. The algorithm takes the following input parameters:

- The Hadoop configuration.
- The `SequenceFile` containing the input `Vectors`.
- The `SequenceFile` containing the initial `Cluster` centers.
- The similarity measure to be used. We'll use `EuclideanDistanceMeasure` as the measure of similarity and experiment with the others later.
- The `convergenceThreshold`. If in an iteration, the centroids don't move more than this distance, no further iterations are done and clustering stops.
- The number of iterations to be done. This is a hard limit; the clustering stops if this threshold is reached.

Mahout algorithms never modify the input directory. This gives you the flexibility to experiment with the algorithm's various parameters. From Java code, you can call the entry point as shown in the following listing to cluster data from the filesystem.

Listing 9.2 The k-means clustering job entry point

```
KmeansDriver.runJob(hadoopConf,
     inputVectorFilesDirPath, clusterCenterFilesDirPath,
     outputDir, new EuclideanDistanceMeasure(),
     convergenceThreshold, numIterations, true, false);
```

TIP Mahout reads and writes data using the Hadoop `FileSystem` class. This provides seamless access to both the local filesystem (via java.io) and distributed filesystems like HDFS and S3FS (using internal Hadoop classes). This way, the same code that works on the local system will also work on the Hadoop filesystem on the cluster, provided the paths to the Hadoop configuration files are correctly set in the environment variables. In Mahout, the bin/mahout shell script finds the Hadoop configuration files automatically from the `$HADOOP_CONF` environment variable.

We'll use the `SparseVectorsFromSequenceFile` tool (discussed earlier in chapter 8) to vectorize documents stored in `SequenceFile` to vectors. Because the k-means algorithm needs the user to input the *k* initial centroids, the MapReduce version similarly needs you to input the path on the filesystem where these *k* centroids are kept. To generate the centroid file you can write some custom logic to select the centroids, as we did in the Hello World example in listing 7.2, or you can let Mahout generate random *k* centroids as detailed next.

RUNNING K-MEANS JOB USING A RANDOM SEED GENERATOR

Let's run k-means clustering over the vectors generated from the Reuters-21578 news collection as described in chapter 8 (section 8.3). In that chapter, the collection was converted to a `Vector` data set and weighted using the TF-IDF measure. The Reuters collection has many topic categories, so you'll set *k* to 20 and see how k-means can cluster the broad topics in the collection.

For running k-means clustering, our mandatory list of parameters includes:

- The Reuters data set in `Vector` format
- The `RandomSeedGenerator` that will seed the initial centroids
- The `SquaredEuclideanDistanceMeasure`
- A large value of `convergenceThreshold` (1.0), because we're using the squared value of the Euclidean distance measure
- The `maxIterations` set to 20
- The number of clusters, *k*, set to 20

If we use the `DictionaryVectorizer` to convert text into vectors with more than one reducer, the data set of vectors in `SequenceFile` format is usually found split into multiple chunks. `KMeansDriver` reads all the files from the input directory assuming they're `SequenceFiles`. So don't worry about the split chunks of vectors, because they'll be taken care of by the Mahout framework.

The same is true for the folder containing the initial centroids. The centroids may be written in multiple `SequenceFile` files, and Mahout reads through all of them. This feature is particularly useful when you have an online clustering system where data is inserted in real time. Instead of appending to the existing file, a new chunk can be created independently and written to without affecting the algorithm.

> **CAUTION** `KMeansDriver` accepts an initial cluster centroid folder as a parameter. It expects a `SequenceFile` full of centroids only if the `-k` parameter isn't set. If the `-k` parameter is specified, the driver class will *erase* the folder and write randomly selected *k* points to a `SequenceFile` there.

`KMeansDriver` is also the main entry point for launching the k-means clustering of the Reuters-21578 news collection. From the Mahout examples directory, execute the Mahout launcher from the shell with `kmeans` as the program name. This driver class will randomly select *k* cluster centroids using `RandomSeedGenerator` and then run the k-means clustering algorithm:

```
$ bin/mahout kmeans -i reuters-vectors/tfidf-vectors/ \
-c reuters-initial-clusters  \
-o reuters-kmeans-clusters \
-dm org.apache.mahout.common.distance.SquaredEuclideanDistanceMeasure \
-cd 1.0 -k 20 -x 20 -cl
```

We're using the Java execution plug-in of the Maven command to execute k-means clustering with the required command-line arguments. The `-k 20` argument specifies that the centroids are randomly generated using `RandomSeedGenerator` and are written to the input clusters folder. The distance measure `SquaredEuclideanDistanceMeasure` need not be mentioned, because it's set by default.

> **TIP** You can see the complete details of the command-line flags and the usage of any Mahout package by setting the `-h` or `--help` command-line flag.

Once the command is executed, clustering iterations will run one by one. Be patient and wait for the centroids to converge. An inspection of Hadoop counters printed at the end of a MapReduce can tell how many of the centroids have converged as specified by the threshold:

```
...
INFO: Counters: 14
May 5, 2010 2:52:35 AM org.apache.hadoop.mapred.Counters log
INFO:   Clustering
May 5, 2010 2:52:35 AM org.apache.hadoop.mapred.Counters log
INFO:     Converged Clusters=6
May 5, 2010 2:52:35 AM org.apache.hadoop.mapred.Counters log
...
```

Had the clustering been done in memory with the preceding parameters, it would have finished in under a minute. The same algorithm over the same data as a MapReduce job takes a couple of minutes. This increase in time is caused by the overhead of the Hadoop library. The library does many checks before starting any map or reduce task, but once they start, Hadoop mappers and reducers run at full speed. This overhead slows down the performance on a single system, but on a cluster, the negative effect of this delay is negated by the reduction in processing time due to the parallelism.

Let's get back to the console where k-means is running. After multiple MapReduce jobs, the k-means clusters converge, the clustering ends, and the points and cluster mappings are written to the output folder.

> **TIP** When you deal with terabytes of data that can't be fit into memory, the MapReduce version of the algorithm is able to scale by keeping the data on the Hadoop Distributed Filesystem (HDFS) and running the algorithm on large clusters. So if your data is small and fits in the RAM, use the in-memory implementation. If your data reaches a point where it can't fit it into memory, start using MapReduce and think of moving the computation to a Hadoop cluster. Check out the Hadoop Quick Start tutorial at http://hadoop.apache.org/common/docs/r0.20.2/quickstart.html to find out more on setting up a pseudo-distributed Hadoop cluster on a Linux box.

The k-means clustering implementation creates two types of directories in the output folder. The clusters-* directories are formed at the end of each iteration: the clusters-0 directory is generated after the first iteration, clusters-1 after the second iteration, and so on. These directories contain information about the clusters: centroid, standard deviation, and so on. The clusteredPoints directory, on the other hand, contains the final mapping from cluster ID to document ID. This data is generated from the output of the last MapReduce operation.

The directory listing of the output folder looks something like this:

```
$ ls -l reuters-kmeans-clusters
drwxr-xr-x  4 user  5000  136 Feb 1 18:56 clusters-0
drwxr-xr-x  4 user  5000  136 Feb 1 18:56 clusters-1
drwxr-xr-x  4 user  5000  136 Feb 1 18:56 clusters-2
...
drwxr-xr-x  4 user  5000  136 Feb 1 18:59 clusteredPoints
```

Once the clustering is done, you need a way to inspect the clusters and see how they're formed. Mahout has a utility called `org.apache.mahout.utils.clustering.ClusterDumper` that can read the output of any clustering algorithm and show the top terms in each cluster and the documents belonging to that cluster. To execute `ClusterDumper`, run the following command:

```
$ bin/mahout clusterdump -dt sequencefile \
-d reuters-vectors/dictionary.file-* \
-s  reuters-kmeans-clusters/clusters-19 -b 10 -n 10
```

The program takes the dictionary file as input. This is used to convert the feature IDs or dimensions of the `Vector` into words.

Running `ClusterDumper` on the output folder corresponding to the last iteration produces output similar to the following:

```
Id: 11736:
    Top Terms: debt, banks, brazil, bank, billion, he, payments, billion
     dlrs, interest, foreign
```

```
Id: 11235:
    Top Terms: amorphous, magnetic, metals, allied signal, 19.39, corrosion,
     allied, molecular, mode, electronic components
...
Id: 20073:
    Top Terms: ibm, computers, computer, att, personal, pc, operating system,
     intel, machines, dos
```

The reason why it's different for different runs is that a random seed generator was used to select the *k* centroids. The output depends heavily on the selection of these centers. In the preceding output, the cluster with ID 11736 has top words like *bank*, *brazil*, *billion*, *debt*, and so on. Most of the articles that belong to this cluster talk about news associated with these words. Note that the cluster with ID 20073 talks about computers and related companies (*ibm*, *att*, *pc*, and so on).

As you can see, you've achieved a decent clustering using a distance measure like `SquaredEuclideanDistanceMeasure`, but it took more than 10 iterations to get there. What's peculiar about text data is that two documents that are similar in content don't necessarily need to have the same length. The Euclidean distance between two documents of different sizes and similar topics is quite large. That is, the Euclidean distance is affected more by the difference in the number of words between the two documents, and less by the words common to both of them. To better understand this, revisit section 7.4.1 and see the Euclidean distance equation's behavior by experimenting with it.

These reasons make Euclidean distance measurement a misfit for text documents. Take a look at this cluster that was created with the Euclidean distance metric:

```
Id: 20978:
    Top Terms: said, he, have, market, would, analysts, he said, from, which,
     has
```

This cluster really doesn't make any sense, especially with words like *said*, *he*, and *the*. To really get good clustering for a given data set, you have to experiment with the different distance measures available in Mahout, as discussed in section 7.4, and see how they perform on the data you have.

You've read that the Cosine distance and Tanimoto measures work well for text documents because they depend more on common words and less on uncommon words. To see this in action, we'll experiment with the Reuters data set and compare the output with the output of the previous clustering. Let's run k-means with `CosineDistanceMeasure`:

```
$ bin/mahout kmeans -i reuters-vectors/tfidf-vectors/ \
-c reuters-initial-clusters  \
-o reuters-kmeans-clusters \
-dm org.apache.mahout.common.distance.CosineDistanceMeasure \
-cd 0.1 -k 20 -x 20 -cl
```

Note that this example sets the convergence threshold to 0.1, instead of using the default value of 0.5, because cosine distances lie between 0 and 1. When the program

runs, one peculiar behavior is noticeable: the clustering speed slows down a bit due to the extra calculation involved in cosine distances, but the clustering converges within a few iterations as compared to over 10 when using the squared Euclidean distance measure. This clearly indicates that cosine distance gives a better notion of similarity between text documents than Euclidean distance.

Once the clustering finishes, we can run the `ClusterDumper` against the output and inspect some of the top words in each cluster. These are some of the interesting clusters:

```
Id: 3475:name:
    Top Terms: iranian, iran, iraq, iraqi, news agency, agency, news, gulf,
     war, offensive
Id: 20861:name:
    Top Terms: crude, barrel, oil, postings, crude oil, 50 cts, effective,
     raises, bbl, cts
```

Experiment with Mahout's k-means algorithm and find the combination of `DistanceMeasure` and `convergenceThreshold` that gives the best clustering for the given problem. Try it on various kinds of data and see how things behave. Explore the various distance measures in Mahout, or try to make one on your own. Though k-means runs impeccably well using randomly seeded clusters, the final centroid locations still depend on their initial positions.

The k-means algorithm is an optimization technique. Given the initial conditions, k-means tries to put the centers at their optimal positions. But it's a greedy optimization that causes it to find the local minima. There can be other centroid positions that satisfy the convergence property, and some of them might be better than the results we just got. We may never find the perfect clusters, but we can apply one powerful technique that will take us closer to it.

9.1.3 *Finding the perfect k using canopy clustering*

For many real-world clustering problems, the number of clusters isn't known beforehand, like the grouping of books in the library example from chapter 7. A class of techniques known as approximate clustering algorithms can estimate the number of clusters as well as the approximate location of the centroids from a given data set. One such algorithm is called canopy generation.

The k-means implementation in Mahout by default generates the `SequenceFile` containing k vectors using the `RandomSeedGenerator` class. Although random centroid generation is fast, there's no guarantee that it will generate good estimates for centroids of the k clusters. Centroid estimation greatly affects the run time of k-means. Good estimates help the algorithm to converge faster and use fewer passes over the data. Automatically finding the number of clusters just by looking at the data is exciting, but the canopy algorithm still has to be told what size clusters to look for, and it will find the number of clusters that have approximately that size.

SEEDING K-MEANS CENTROIDS USING CANOPY GENERATION

Canopy generation, also known as *canopy clustering*, is a fast approximate clustering technique. It's used to divide the input set of points into overlapping clusters known as canopies. The word canopy, in this context, refers to an enclosure of points, or a cluster. Canopy clustering tries to estimate the approximate cluster centroids (or canopy centroids) using two distance thresholds.

Canopy clustering's strength lies in its ability to create clusters extremely quickly—it can do this with a single pass over the data. But its strength is also its weakness. This algorithm may not give accurate and precise clusters. But it can give the optimal number of clusters without even specifying the number of clusters, *k*, as required by k-means.

The algorithm uses a fast distance measure and two distance thresholds, T_1 and T_2, with $T_1 > T_2$. It begins with a data set of points and an empty list of canopies, and then iterates over the data set, creating canopies in the process. During each iteration, it removes a point from the data set and adds a canopy to the list with that point as the center. It loops through the rest of the points one by one. For each one, it calculates the distances to all the canopy centers in the list. If the distance between the point and any canopy center is within T_1, it's added into that canopy. If the distance is within T_2, it's removed from the list and thereby prevented from forming a new canopy in the subsequent loops. It repeats this process until the list is empty.

This approach prevents all points close to an already existing canopy (distance $< T_2$) from being the center of a new canopy. It's detrimental to form another redundant canopy in close proximity to an existing one. Figure 9.3 illustrates the canopies created using this method. Which clusters are formed depends only on the choice of distance thresholds.

UNDERSTANDING THE CANOPY GENERATION ALGORITHM

The canopy generation algorithm is executed using the `CanopyClusterer` or the `CanopyDriver` class. The former does in-memory clustering of the points, whereas the latter implements it as MapReduce jobs. These jobs can be run like a regular Java program and can read and write data from the disk. They can also be run on a Hadoop cluster, reading data from and writing it to a distributed filesystem.

We're going to use the same random point generator function as earlier to create vectors scattered in the 2-dimensional plane, like in a normal distribution. For this example, we'll generate three normal distributions. Listing 9.3 runs the in-memory version of canopy clustering using the `CanopyClusterer` with the following parameters:

- The input `Vector` data is in `List<Vector>` format.
- The `DistanceMeasure` is `EuclideanDistanceMeasure`.
- The value of T_1 is 3.0.
- The value of T_2 is 1.5.

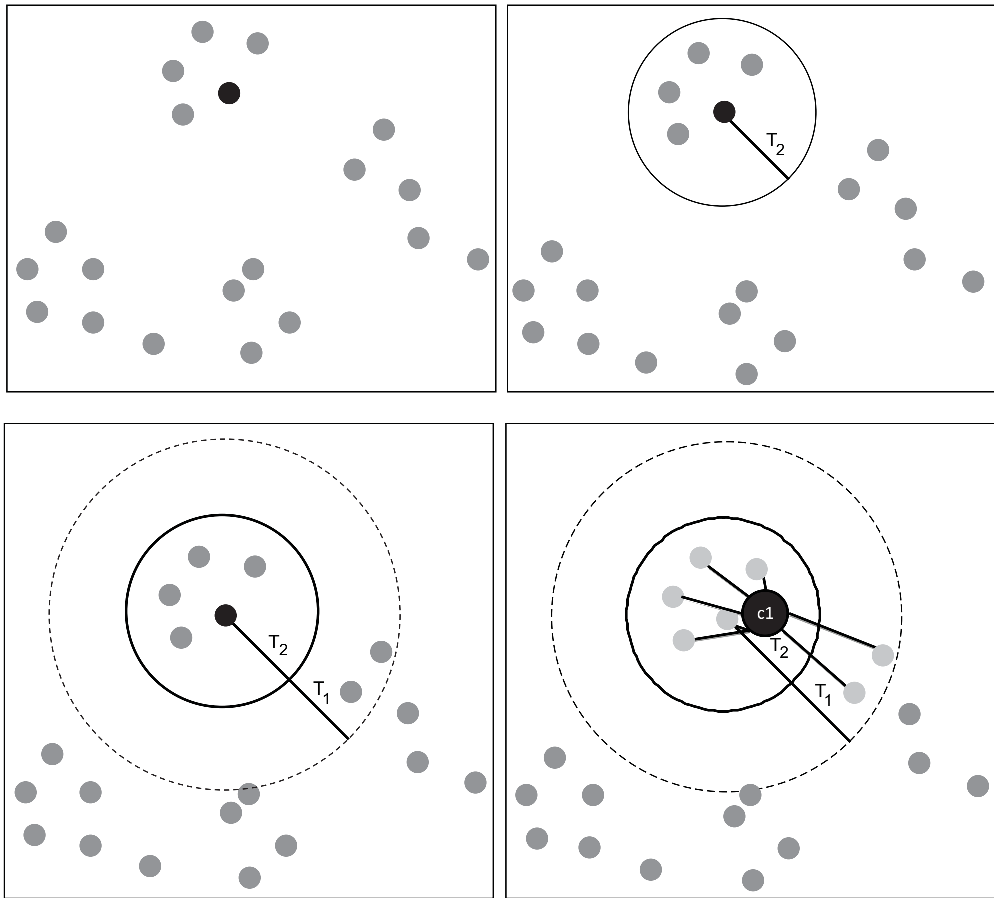


Figure 9.3 Canopy clustering: if you start with a point (top left) and mark it as part of a canopy, all the points within distance T_2 (top right) are removed from the data set and prevented from becoming new canopies. The points within the outer circle (bottom-right) are also put in the same canopy, but they're allowed to be part of other canopies. This assignment process is done in a single pass on a mapper. The reducer computes the average of the centroid (bottom right) and merges close canopies.

Listing 9.3 In-memory example of the canopy generation algorithm

```
public static void CanopyExample() {
  List<Vector> sampleData = new ArrayList<Vector>();

  generateSamples(sampleData, 400, 1, 1, 2);
  generateSamples(sampleData, 300, 1, 0, 0.5);
  generateSamples(sampleData, 300, 0, 2, 0.1);

  List<Canopy> canopies = CanopyClusterer.createCanopies(
    sampleData, new EuclideanDistanceMeasure(), 3.0, 1.5);

  for(Canopy canopy : canopies) {
    System.out.println("Canopy id: " + canopy.getId()
```

Generate 3 sets of points

Run CanopyClusterer

```
                    + " center: " +
                    canopy.getCenter().asFormatString());      <- Read center of canopy and print
  }
 }
```

The `DisplayCanopy` class in Mahout's `mahout-examples` module displays a set of points in a 2-dimensional plane and shows how the canopy generation is done using the in-memory `CanopyClusterer`. Typical output of `DisplayCanopy` is shown in figure 9.4.

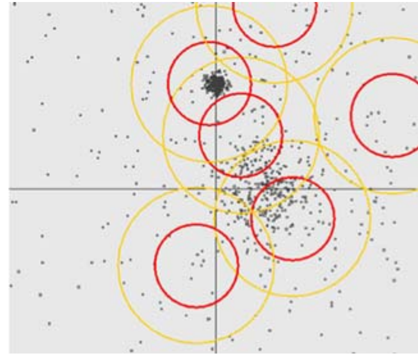


Figure 9.4 An example of in-memory canopy generation visualized using the `DisplayCanopy` class, that's included in examples bundled with Mahout. We started with T_1=3.0 and T_2=1.5 and clustered the randomly generated points.

Canopy clustering doesn't require you to specify the number of cluster centroids as a parameter. The number of centroids formed depends only on the choice of the distance measures, T_1 and T_2. The in-memory canopy clustering implementation works with a list of `Vector` objects, just like the k-means implementation. If the data set is huge, it's not possible to run this algorithm on a single machine, and the MapReduce job is required. The MapReduce version of the canopy clustering implementation does a slight approximation as compared to the in-memory one and produces a slightly different set of canopies for the same input data. This is nothing to be alarmed about when the data is huge. The output of canopy clustering is a great starting point for k-means, which improves clustering due to the increased precision of the initial centroids as compared to random selection.

Using the canopies you've generated, you can assign points to the nearest canopy centers, in theory clustering the set of points. This is called *canopy clustering* instead of *canopy generation*. In Mahout, the `CanopyDriver` class does canopy centroid generation and optionally also clusters the points if the `runClustering` parameter is set to `true`.

Next, you'll run canopy generation on the Reuters collection and figure out the value of *k*.

RUNNING THE CANOPY GENERATION ALGORITHM TO SELECT *k* CENTROIDS

Let's now generate canopy centroids from the Reuters `Vector` data set. To generate the centroids, you'll set the distance measure to `EuclideanDistanceMeasure` and use threshold values of `t1=2000` and `t2=1500`. Remember that the Euclidean distance measure gives very large distance values for sparse document vectors, so large values for `t1` and `t2` are necessary to get meaningful clusters.

The distance threshold values (`t1` and `t2`) selected in this case produce less than 50 centroid points for the Reuters collection. We estimated these threshold values after running the `CanopyDriver` multiple times over the input data. Due to the fast nature of canopy clustering, it's possible to experiment with various parameters and see the results much more quickly than when using expensive techniques like k-means.

To run canopy generation over the Reuters data set, execute the `canopy` program using the Mahout launcher as follows:

```
$ bin/mahout canopy -i reuters-vectors/tfidf-vectors \
-o reuters-canopy-centroids \
-dm org.apache.mahout.common.distance.EuclideanDistanceMeasure \
-t1 1500 -t2 2000
```

Within a minute, `CanopyDriver` will generate the centroids in the output folder. You can inspect the canopy centroids using the cluster dumper utility, as you did for k-means earlier in this chapter.

Next, we'll use this set of centroids to improve k-means clustering.

IMPROVING K-MEANS CLUSTERING USING CANOPY CENTERS

We can now run the k-means clustering algorithm using the canopy centroids generated in the previous section. For that, all we need to do is set the clusters parameter (`-c`) to the canopy clustering output folder and remove the `-k` command-line parameter in `KMeansDriver`. (Be careful: if the `-k` flag is set, the `RandomSeedGenerator` will overwrite the canopy centroid folder.)

We'll use `TanimotoDistanceMeasure` in k-means to get clusters as follows:

```
$ bin/mahout kmeans -i reuters-vectors/tfidf-vectors \
-o reuters-kmeans-clusters \
-dm org.apache.mahout.common.distance.TanimotoDistanceMeasure  \
-c reuters-canopy-centroids/clusters-0 -cd 0.1 -ow -x 20 -cl
```

After the clustering is done, use `ClusterDumper` to inspect the clusters. Some of them are listed here:

```
Id: 21523:name:
    Top Terms:
tones, wheat, grain, said, usda, corn, us, sugar, export, agriculture
Id: 21409:name:
    Top Terms:
stock, share, shares, shareholders, dividend, said, its, common, board,
     company
Id: 21155:name:
    Top Terms:
oil, effective, crude, raises, prices, barrel, price, cts, said, dlrs
Id: 19658:name:
    Top Terms:
drug, said, aids, inc, company, its, patent, test, products, food
Id: 21323:name:
    Top Terms:
       7-apr-1987, 11, 10, 12, 07, 09, 15, 16, 02, 17
```

Note the last cluster. Although the others seem to be great topic groups, the last one looks meaningless. Nevertheless, the clustering grouped documents with these terms because the terms occur together in these documents. Another issue is that words like *its* and *said* in these clusters are useless from a language standpoint. The algorithm simply doesn't know that. Any clustering algorithm can generate good clustering provided the highest weighted features of the vector represent good features of the document.

In sections 8.3 and 8.4, you saw how TF-IDF and normalization gave higher weights to the important features and lower weights to the stop-words, but occasionally spurious clusters do surface. A quick and effective way to avoid this problem is to remove these words from ever occurring as features in the document `Vector`. In the next case study, you'll see how to fix this by using a custom Lucene `Analyzer` class.

Canopy clustering is a good approximate clustering technique, but it suffers from memory problems. If the distance thresholds are close, too many canopies are generated, and this increases memory usage in the mapper. This might exceed available memory while running on a large data set with a bad set of thresholds. The parameters need to be tuned to fit the data set and the clustering problem, as you'll see next.

The next example we'll look at creates a clustering module for a news website. We chose a news website because it best represents a dynamic system where content needs to be organized with very good precision. Clustering can help solve the issues related to such content systems.

9.1.4 Case study: clustering news articles using k-means

In this case study, we're going to assume that we're in charge of a fictional news aggregation website called AllMyNews.com. A user who comes to the website uses keywords to search for the content they're looking for. If they see an interesting article, they can use the words in the article to search for related articles, or they can drill down into the news category and explore related news articles there. Usually we rely on human editors to find related items and help categorize and cross-link the whole website. But if articles are coming in at tens of thousands per day, human intervention might prove too expensive. Enter clustering.

Using clustering, we may be able to find related stories automatically and be able to give the user a better browsing experience. In this section, you'll use k-means clustering to implement such a feature. Look at figure 9.5 for an example of what the feature should look like in practice. For a news story on the website, we'll show the user a list of all related news articles.

For any given article, we can store the cluster in which the articles reside. When a user requests articles related to the one they're reading, we'll pick out all the articles in the cluster, sort them based on the distance to the given article, and present them to the user.

Figure 9.5 An example of related-articles functionality taken from the Google News website. The links to similar stories within the cluster are shown at the bottom in bold. The top related articles are shown as links above that.

That's a great initial design for a news-clustering system, but it doesn't solve all the issues completely. Let's list some real-life problems we might face in such a context:

- There are articles coming in every minute, and the website needs to refresh its clusters and indexes.
- There might be multiple stories breaking out at the same time, so we'll require separate clusters for them, which means we need to add more centroids every time this happens.
- The quality of the text content is questionable, because there are multiple sources feeding in the data. We need to have mechanisms to clean up the content when doing feature selection.

We'll start with an efficient k-means clustering implementation to cluster the news articles offline. Here, the word *offline* means we'll write the documents into `SequenceFiles` and start the clustering as a backend process. We won't go into the details of how the news data is stored. We'll assume for simplicity that document storage and retrieval blocks can't be replaced easily by database read/write code.

> **NOTE** In the coming chapters, we'll modify this case study and add advanced techniques using Mahout to solve issues related to speed and quality. Finally, in chapter 12 we'll show a working, tuned, and scalable clustering example applied to some real-world datasets that can be adapted for different applications.

Listing 9.4 shows the code that clusters news articles from `SequenceFiles`, and listing 9.5 shows a custom Lucene `Analyzer` class that prunes away non-alphabetic features from the data.

Listing 9.4 News clustering using canopy generation and k-means clustering

```
public class NewsKMeansClustering {
  public static void main(String args[]) throws Exception {
    int minSupport = 2;
    int minDf = 5;
    int maxDFPercent = 95;
    int maxNGramSize = 2;
    int minLLRValue = 50;
    int reduceTasks = 1;
    int chunkSize = 200;
    int norm = 2;
    boolean sequentialAccessOutput = true;

    String inputDir = "inputDir";

    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(conf);

    String outputDir = "newsClusters";
    HadoopUtil.delete(new Path(outputDir));
    Path tokenizedPath = new Path(outputDir,
        DocumentProcessor.TOKENIZED_DOCUMENT_OUTPUT_FOLDER);
    MyAnalyzer analyzer = new MyAnalyzer();          <- Custom Lucene Analyzer
```

```
    DocumentProcessor.tokenizeDocuments(new Path(inputDir),
      analyzer.getClass().asSubclass(Analyzer.class),
      tokenizedPath, conf);                              <- Tokenize text

    DictionaryVectorizer.createTermFrequencyVectors(tokenizedPath,
      new Path(outputDir), conf, minSupport, maxNGramSize, minLLRValue,
      2, true, reduceTasks,
      chunkSize, sequentialAccessOutput, false);
    TFIDFConverter.processTfIdf(
      new Path(outputDir ,
     DictionaryVectorizer.DOCUMENT_VECTOR_OUTPUT_FOLDER),
      new Path(outputDir), conf, chunkSize, minDf,
      maxDFPercent, norm, true, sequentialAccessOutput, false,
     reduceTasks);                                       <- Calculate TF-IDF vectors
    Path vectorsFolder = new Path(outputDir, "tfidf-vectors");
    Path canopyCentroids = new Path(outputDir ,
                                   "canopy-centroids");
    Path clusterOutput = new Path(outputDir , "clusters");

    CanopyDriver.run(vectorsFolder, canopyCentroids,     <- Run canopy centroid generation
      new EuclideanDistanceMeasure(), 250, 120,
      false, false);
    KMeansDriver.run(conf, vectorsFolder,                <- Run k-means algorithm
      new Path(canopyCentroids, "clusters-0"),
      clusterOutput, new TanimotoDistanceMeasure(), 0.01,
      20, true, false);

    SequenceFile.Reader reader = new SequenceFile.Reader(fs,
        new Path(clusterOutput
            + Cluster.CLUSTERED_POINTS_DIR + "/part-00000"), conf);

    IntWritable key = new IntWritable();
    WeightedVectorWritable value = new WeightedVectorWritable();
    while (reader.next(key, value)) {                    <- Read Vector to Cluster mappings
       System.out.println(key.toString() + " belongs to cluster "
       + value.toString());
    }
    reader.close();
  }
}
```

Listing 9.5 A custom Lucene analyzer that filters non-alphabetic tokens

```
public class MyAnalyzer extends Analyzer {

  private final Pattern alphabets = Pattern.compile("[a-z]+");

  @Override
  public TokenStream tokenStream(String fieldName, Reader reader) {
    TokenStream result = new StandardTokenizer(
        Version.LUCENE_CURRENT, reader);
    result = new StandardFilter(result);
    result = new LowerCaseFilter(result);
    result = new StopFilter(true, result,
                    StandardAnalyzer.STOP_WORDS_SET);    <- Use Lucene filters

    TermAttribute termAtt =
       (TermAttribute) result.addAttribute(
```

```
          TermAttribute.class);
    StringBuilder buf = new StringBuilder();
    try {
      while (result.incrementToken()) {
        if (termAtt.termLength() < 3) continue;          // Filter small tokens
        String word = new String(
            termAtt.termBuffer(), 0, termAtt.termLength());
        Matcher m = alphabets.matcher(word);

        if (m.matches()) {                                // Filter non-alphabetic tokens
          buf.append(word).append(" ");
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    }

    return new WhiteSpaceTokenizer(new StringReader(buf.toString()));
  }
}
```

This `NewsKMeansClustering` example is straightforward. The documents are stored in the input directory. From these, we create vectors from unigrams and bigrams that contain only alphabetic characters. Using the generated `Vectors` as input, we run the canopy centroid generation job to create the seed set of centroids for the k-means clustering algorithm. Finally, at the end of k-means clustering, we read the output and save it to the database.

The next section looks at the other algorithms in Mahout that can take us further than k-means.

9.2 Beyond k-means: an overview of clustering techniques

K-means produces rigid clustering. For example, a news article that talks about the influence of politics in biotechnology could be clustered either with the politics documents or with the biotechnology documents, but not with both. Because we're trying to tune the related-articles feature, we might need overlapping or fuzzy information. We might also need to model the point distribution of our data, and this isn't something k-means was designed to do.

K-means is just one type of clustering. There are many other clustering algorithms designed on different principles, which we'll look at next.

9.2.1 Different kinds of clustering problems

Recall that clustering is simply a process of putting things into groups. To do more than simple grouping, you need to understand the different kinds of problems in clustering. These problems and their solutions fall mainly into four categories:

- Exclusive clustering
- Overlapping clustering
- Hierarchical clustering
- Probabilistic clustering

Let's look at each in turn.

EXCLUSIVE CLUSTERING

In exclusive clustering, an item belongs exclusively to one cluster, not several. Recall the discussion about clustering books in a library from chapter 7. We could have simply associated a book like *Harry Potter* only with the cluster of fiction books. Thus, *Harry Potter* would have exclusively belonged to the fiction cluster. K-means does this sort of exclusive clustering, so if the clustering problem demands this behavior, k-means will usually do the trick.

OVERLAPPING CLUSTERING

What if we wanted to do non-exclusive clustering; that is, put *Harry Potter* not only in fiction but also in a young adult cluster as well as under fantasy. An overlapping clustering algorithm like fuzzy k-means achieves this easily. Moreover, fuzzy k-means also indicates the degree with which an object is associated with a cluster. *Harry Potter* might be inclined more towards the fantasy cluster than the young adult cluster. The difference between exclusive and overlapping clustering is illustrated in figure 9.6.

HIERARCHICAL CLUSTERING

Now, assume a situation where we have two clusters of books, one for fantasy and the other for space travel. *Harry Potter* is in the cluster of fantasy books, but these two clusters, space travel and fantasy, could be visualized as subclusters of fiction. Hence, we can construct a fiction cluster by merging these and other similar clusters. The fiction and fantasy clusters have a parent-child relationship, as illustrated in figure 9.7, and hence the name *hierarchical clustering*.

Similarly, we could keep grouping clusters into bigger and bigger ones. At a certain point, the clusters would be so large and so generic that they'd be useless as

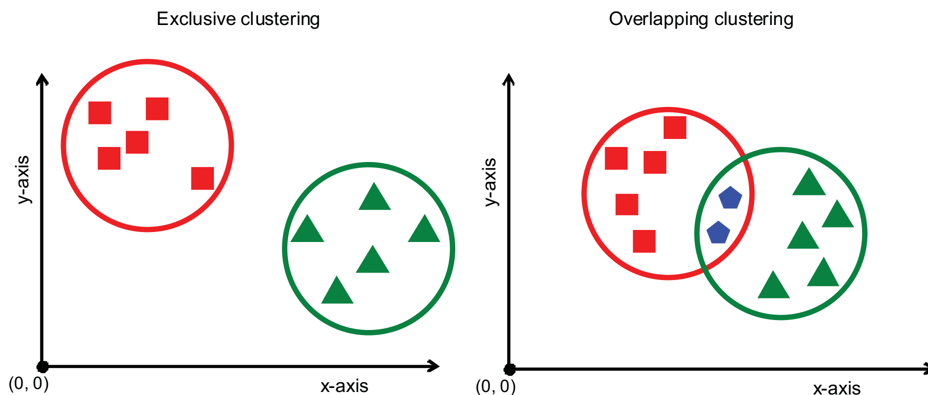


Figure 9.6 Exclusive clustering versus overlapping clustering with two centers. In the former, squares and triangles each have their own cluster, and each belongs only to one cluster. In overlapping clustering, some shapes, like pentagons, can belong to both clusters with some probability, so they're part of both clusters instead of having a cluster of their own.

groupings. Nevertheless, this is a useful method of clustering: merging small clusters until it becomes undesirable to do so. Methods that uncover such a systematic tree-like hierarchy for a given data collection are called hierarchical clustering algorithms.

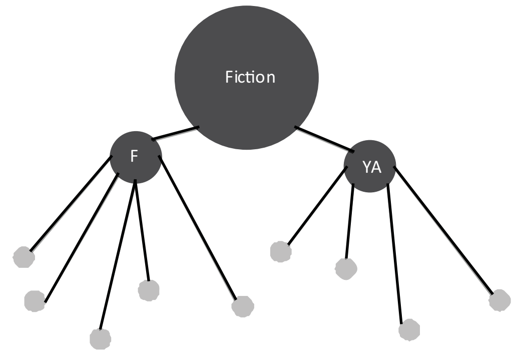


Figure 9.7 Hierarchical clustering: a bigger cluster groups two or more smaller clusters in the form of a tree-like hierarchy. Recall the library example in chapter 7—we did a crude form of hierarchical clustering when we simply stacked books based on the similarity between them.

PROBABILISTIC CLUSTERING

A probabilistic model is usually a characteristic shape or a type of distribution of a set of points in an *n*-dimensional plane. There are a variety of probabilistic models that fit known data patterns. Probabilistic clustering algorithms try to fit a probabilistic model over a data set and adjust the model's parameters to correctly fit the data. Such correct fits rarely happen; instead, these algorithms give a percentage match or a probability value, which indicates how well the model fits the cluster.

To explain how this fitting happens, look at the 2-dimensional example in figure 9.8. Say that we somehow know that all points in a plane are distributed in various regions with an elliptical shape, but we don't know the center and radius or axes of these regions. We'll choose an elliptical model and try to fit it to the data; we'll move, stretch, or contract each ellipse to best fit a region and do this for all the regions. This is called model-based clustering.

A typical example of this type is the Dirichlet clustering algorithm, which does the fitting based on a model provided by the user. We'll see this clustering algorithm in action in section 9.4. Before we get there, though, you need to understand how different clustering algorithms are grouped based on their strategies.

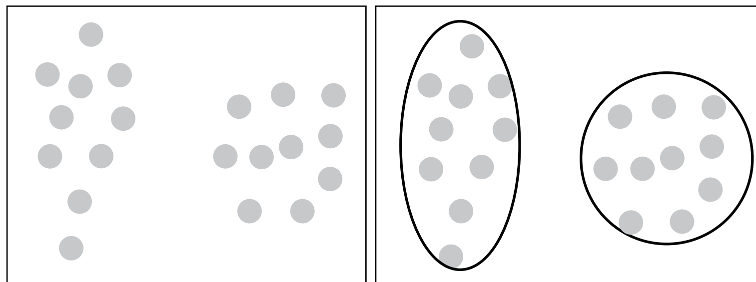


Figure 9.8 A simplified view of probabilistic clustering. The initial set of points is on the left. On the right, the first set of points matches an elongated elliptical model whereas the second one is more symmetric.

9.2.2 Different clustering approaches

Different clustering algorithms take different approaches. We can categorize most of these approaches as follows:

- Fixed number of centers
- Bottom-up approach
- Top-down approach

There are many other clustering algorithms that have unique ways of clustering, but you may never encounter them in Mahout, because they currently aren't scalable on large data sets. Instead, we'll explore the algorithms in those three categories.

FIXED NUMBER OF CENTERS

These clustering methods fix the number of clusters ahead of time. The count of clusters is typically denoted by the letter *k*, which originated from the *k* of the k-means algorithm, which is the best known of these methods. The idea is to start with *k* and to modify these *k* cluster centers to better fit the data. Once the centers have converged to match the data, the points in the data set are assigned to the centroid closest to them.

Fuzzy k-means is another example of an algorithm that requires a fixed number of clusters. Unlike k-means, which does exclusive clustering, fuzzy k-means does overlapping clustering.

BOTTOM-UP APPROACH: FROM POINTS TO CLUSTERS VIA GROUPING

When you have a set of points in *n*-dimensions, you can do two things: assume that all points belong to a single cluster and start dividing the cluster into smaller clusters, or assume that each of the data point begins in its own cluster and start grouping them iteratively. The former is a *top-down approach* and the latter is a *bottom-up approach*.

The bottom-up clustering algorithms work as follows: from a set of points in an *n*-dimensional space, the algorithm finds the pairs of points close to each other and merges them into one cluster, as illustrated in figure 9.9. This merge is done only if the distance between them is below a certain threshold value. If not, those points are

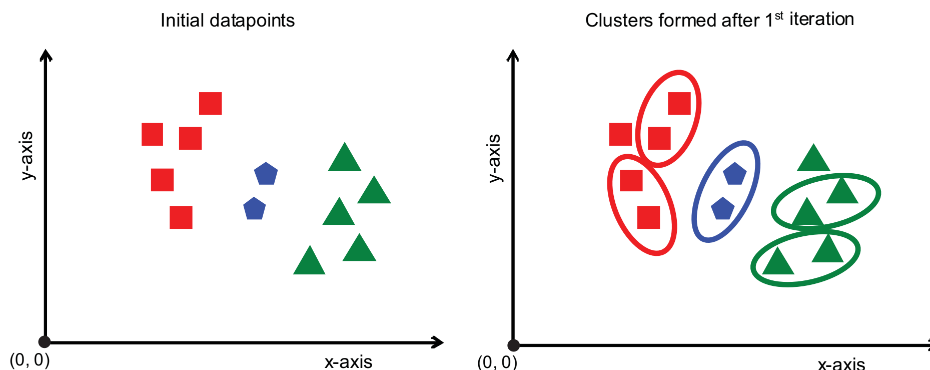


Figure 9.9 Bottom-up clustering: After every iteration the clusters are merged to produce larger and larger clusters until it's not feasible to merge based on the given distance measure.

left alone. This process of merging the clusters using the distance measure is repeated until nothing can be merged anymore.

TOP-DOWN APPROACH: SPLITTING THE GIANT CLUSTER

In the top-down approach, you start with assigning all points to a single giant cluster. Then you find the best possible way to split this giant cluster into two smaller clusters, as illustrated in figure 9.10. These clusters are divided repeatedly until you get clusters that are meaningful, based on some distance measure criterion.

Though this is straightforward, finding the best possible split for a set of *n*-dimensional points isn't easy. Moreover, most of these algorithms can't be easily reduced into the MapReduce form, so they aren't currently available in Mahout.

An example of a top-down algorithm is spectral clustering. In spectral clustering, the splitting is decided by finding the line or plane that cuts the data into two sets with a larger margin between the two sets.

ADVANTAGES AND DISADVANTAGES OF THE TOP-DOWN AND BOTTOM-UP APPROACHES

The beauty of the top-down and bottom-up approaches is that they don't require the user to input the cluster size. This means that in a data set where the distribution of the points is unclear, both these types of algorithms output clusters based solely on the similarity metric. This works quite well in many applications.

These approaches are still being researched and they can't be run as MapReduce jobs. But even though Mahout has no implementations of these methods, other specialized algorithms implemented in Mahout can run as MapReduce jobs without specifying the number of clusters.

The lack of hierarchical clustering algorithms in MapReduce is easily circumvented by the smart use of k-means, fuzzy k-means, and Dirichlet clustering. To get the hierarchy, start with a small number of clusters (k) and repeat clustering with increasing values of k. Alternatively, you can start with a large number of centroids and start clustering the cluster centroids with decreasing values of k. This mimics the

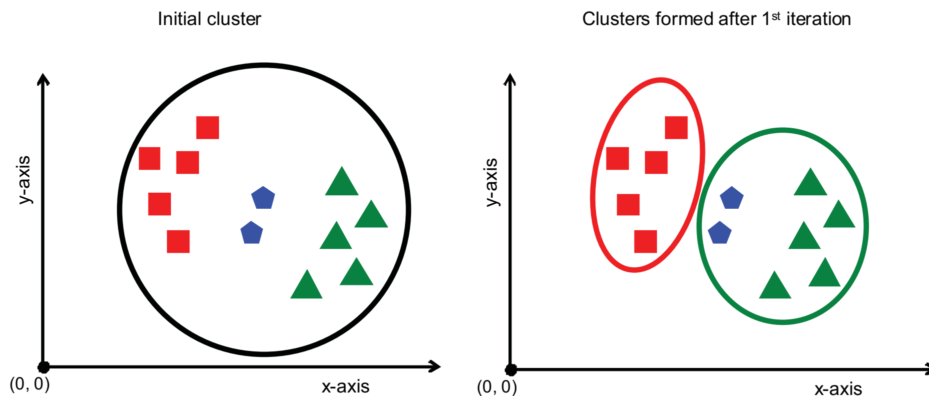


Figure 9.10 Top-down clustering: during each iteration, the clusters are divided into two by finding the best splitting until you get the clusters you desire.

hierarchical clustering behavior while making full use of the scalable nature of Mahout implementations.

The next section deals with the fuzzy k-means algorithm in detail. We use it to improve our clustering of related articles for our news website, AllMyNews.com.

9.3 Fuzzy k-means clustering

As the name says, the fuzzy k-means clustering algorithm does a fuzzy form of k-means clustering. Instead of the exclusive clustering in k-means, fuzzy k-means tries to generate overlapping clusters from the data set. In the academic community, it's also known as the *fuzzy c-means* algorithm. You can think of it as an extension of k-means.

K-means tries to find the hard clusters (where each point belongs to one cluster) whereas fuzzy k-means discovers the soft clusters. In a soft cluster, any point can belong to more than one cluster with a certain affinity value towards each. This affinity is proportional to the distance from the point to the centroid of the cluster. Like k-means, fuzzy k-means works on those objects that can be represented in *n*-dimensional vector space and it has a distance measure defined.

9.3.1 Running fuzzy k-means clustering

The fuzzy k-means algorithm is available in the `FuzzyKMeansClusterer` and `FuzzyKMeansDriver` classes. The former is an in-memory implementation, and the latter uses MapReduce.

Let's look at an example. You'll use the same random point generator function you used earlier to scatter points in a 2-dimensional plane. Listing 9.6 shows the in-memory implementation using `FuzzyKMeansClusterer` with the following parameters:

- The input `Vector` data is in `List<Vector>` format.
- The `DistanceMeasure` is `EuclideanDistanceMeasure`.
- The threshold of convergence is 0.01.
- The number of clusters, *k*, is 3.
- The fuzziness parameter, *m*, is 3. (This parameter will be explained in section 9.3.2.)

Listing 9.6 In-memory example of fuzzy k-means clustering

```
public static void FuzzyKMeansExample() {
  List<Vector> sampleData = new ArrayList<Vector>();

  generateSamples(sampleData, 400, 1, 1, 3);
  generateSamples(sampleData, 300, 1, 0, 0.5);
  generateSamples(sampleData, 300, 0, 2, 0.1);

   int k = 3;
  List<Vector> randomPoints
     = RandomPointsUtil.chooseRandomPoints(sampleData, k);
     List<SoftCluster> clusters = new ArrayList<SoftCluster>();

     int clusterId = 0;
```

Generate three sets of points

```
    for (Vector v : randomPoints) {
      clusters.add(new SoftCluster(v, clusterId++,
          new EuclideanDistanceMeasure()));
    }

    List<List<SoftCluster>> finalClusters
        = FuzzyKMeansClusterer.clusterPoints(sampleData,
            clusters, new EuclideanDistanceMeasure(),
            0.01, 3, 10);                                    <-- Run FuzzyKMeansClusterer
    for(SoftCluster cluster : finalClusters.get(
        finalClusters.size() - 1)) {
   System.out.println("Fuzzy Cluster id: "
       + cluster.getId()
       + " center: " + cluster.getCenter().asFormatString());   <-- Read centers of fuzzy clusters
    }

  }
```

The DisplayFuzzyKMeans class in Mahout's mahout-examples module is a good tool for visualizing this algorithm on a 2-dimensional plane. DisplayFuzzy-KMeans runs as a Java Swing application and produces output like that shown in figure 9.11.

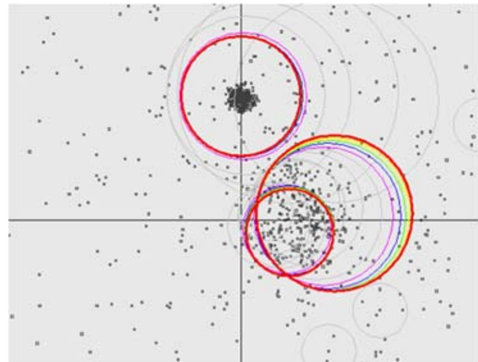


Figure 9.11 Fuzzy k-means clustering: the clusters look like they're overlapping each other, and the degree of overlap is decided by the fuzziness parameter.

MAPREDUCE IMPLEMENTATION OF FUZZY K-MEANS

The MapReduce implementation of fuzzy k-means looks similar to that of the k-means implementation in Mahout. These are the important parameters:

- The input data set in Vector format.
- The RandomSeedGenerator to seed the initial *k* clusters.
- The distance measure: SquaredEuclideanDistanceMeasure in this case.
- A large value of convergenceThreshold, such as -cd 1.0, because we're using the squared value of the distance measure.
- A value for maxIterations; we'll use the default value of -x 10.
- The coefficient of normalization, or the fuzziness factor, with a value greater than -m 1.0; this will be explained in the section 9.3.2.

To run the fuzzy k-means clustering over the input data, use the Mahout launcher with the fkmeans program name as follows:

```
$ bin/mahout fkmeans \
-i reuters-vectors/tfidf-vectors/ -c reuters-fkmeans-centroids \
-o reuters-fkmeans-clusters -cd 1.0 -k 21 -m 2 -ow -x 10\
-dm org.apache.mahout.common.distance.SquaredEuclideanDistanceMeasure
```

As with k-means, `FuzzyKMeansDriver` will automatically run the `RandomSeedGenerator` if the number of clusters (`-k`) flag is set. Once the random centroids are generated, fuzzy k-means clustering will use them as the input set of *k* centroids. The algorithm runs multiple iterations over the data set until the centroids converge, each time creating the output in the cluster- folder. Finally, it runs another job that generates the probabilities of each point belonging to a particular cluster based on the distance measure and the fuzziness parameter (`-m`).

Before we get into the details of the fuzziness parameter, it's a good idea to inspect the clusters using the `ClusterDumper` tool. `ClusterDumper` shows the top words of the cluster for each centroid. To get the actual mapping of the points to the clusters, you need to read the `SequenceFiles` in the clusteredPoints/ folder. Each entry in the sequence file has a key, which is the identifier of the vector, and a value, which is the list of cluster centroids with an associated numerical value, which indicates how well the point belongs to that particular centroid.

9.3.2 How fuzzy is too fuzzy?

Fuzzy k-means has a parameter, *m*, called the fuzziness factor. Like k-means, fuzzy k-means loops over the data set but instead of assigning vectors to the nearest centroids, it calculates the degree of association of the point to each of the clusters.

Suppose for a vector, V, that d_1, d_2, ... d_k are the distances to each of the *k* cluster centroids. The degree of association (`u1`) of vector (V) to the first cluster (C_1) is calculated as

$$u_i = \frac{1}{\left(\frac{d_1}{d_1}\right)^{\frac{2}{m-1}} + \left(\frac{d_1}{d_2}\right)^{\frac{2}{m-1}} + \ldots + \left(\frac{d_1}{d_k}\right)^{\frac{2}{m-1}}}$$

Similarly, you can calculate the degree of association to other clusters by replacing d_1 in the numerators of the denominator expression with d_2, d_3, and so on. It's clear from the expression that *m* should be greater than 1, or else the denominator of the fraction becomes 0 and things break down.

If you choose a value of 2 for *m*, you'll see that all degrees of association for any point sum up to one. If, on the other hand, *m* comes very close to 1, like 1.000001, more importance will be given to the centroid closest to the vector. The fuzzy k-means algorithm starts behaving more like the k-means algorithm as *m* gets closer to 1. If *m* increases, the fuzziness of the algorithm increases, and you'll begin to see more and more overlap.

The fuzzy k-means algorithm also converges better and faster than the standard k-means algorithm.

9.3.3 Case study: clustering news articles using fuzzy k-means

The related-articles functionality will certainly be richer with partial overlap of clusters. The partial score will help us rank the related articles according to their relatedness to the cluster. In the following listing, the case study example from section 9.1.4 is modified to use the fuzzy k-means algorithm and retrieve the fuzzy cluster membership information.

Listing 9.7 News clustering using fuzzy k-means clustering

```
public class NewsFuzzyKMeansClustering {
  public static void main(String args[]) throws Exception {
    ...

    String vectorsFolder = outputDir + "/tfidf-vectors";
    String canopyCentroids = outputDir + "/canopy-centroids";
    String clusterOutput = outputDir + "/clusters/";

    CanopyDriver.run(conf, new Path(vectorsFolder), new Path(
        canopyCentroids), new ManhattanDistanceMeasure(), 3000.0,
      2000.0, false, false);

    FuzzyKMeansDriver.run(conf, new Path(vectorsFolder), new Path(
        canopyCentroids, "clusters-0"), new Path(clusterOutput),
      new TanimotoDistanceMeasure(), 0.01, 20, 2.0f, true, true, 0.0,
      false);

    SequenceFile.Reader reader = new SequenceFile.Reader(fs,
        new Path(clusterOutput + Cluster.CLUSTERED_POINTS_DIR
                 + "/part-m-00000"), conf);

    IntWritable key = new IntWritable();
    WeightedVectorWritable value = new WeightedVectorWritable();
    while (reader.next(key, value)) {
      System.out.println("Cluster: " + key.toString()
        + " " + value.getVector().asFormatString());
      // Write code here to save the cluster mapping to our database
    }
    reader.close();
  }
}
```

Run canopy generation job

Run fuzzy k-means clustering

Read mapping from output

Print clusters and probabilities

The fuzzy k-means algorithm gives us a way to refine the related-articles code. Now we know by what degree a point belongs to a cluster. Using this information, we can find the top clusters a point belongs to, and use the degree to find the weighted score of articles. This way we avoid the strictness of exclusive clustering and identify better-related articles for documents lying on the boundaries of a cluster.

9.4 Model-based clustering

The complexities of clustering algorithms have increased progressively in this chapter. We started with k-means, a fast clustering algorithm. Then we captured partial clustering membership using fuzzy k-means. We also optimized the clustering using centroid generation algorithms like canopy clustering.

What more do we want to know about these clusters? To better understand the structures within the data, we may need a method that's completely different from the algorithms we've discussed so far—model-based clustering methods.

Before we focus on what model-based clustering is, we need to look at some of the issues faced by k-means and other related algorithms. One such method is Dirichlet process clustering, and we'll be dealing with it throughout this section. But to understand it, you need to first understand what kind of problems it solves, and the best way

to do that is to compare it to the clustering algorithms you've already seen in this chapter. So we'll look at what k-means can't do, and then see how Dirichlet process clustering solves that problem.

9.4.1 Deficiencies of k-means

Say you wanted to cluster a data set into *k* clusters. You've learned how to run k-means and get the clusters quickly. K-means works well because it can easily divide clusters using a linear distance.

What if you knew that the clusters were based on a normal distribution and are mixed together and overlap each other? Here, you might be better off with fuzzy k-means clustering.

What if the clusters themselves aren't in a normal distribution? What if the clusters have an ovoid shape? Neither k-means nor fuzzy k-means knows how to use this information to improve the clustering. Before we look at how to solve this problem, let's first look at an example where k-means clustering fails to describe a simple distribution of data.

ASYMMETRICAL NORMAL DISTRIBUTION

You're going to run k-means clustering using points generated from an asymmetrical normal distribution. What this means is that instead of the points being scattered in two dimensions around a point in a circular area, you're going to make the point-generator generate clusters of points having different standard deviations in different directions. This creates an ellipsoidal area in which the points are concentrated. We'll then run the in-memory k-means implementation over this data.

Figure 9.12 shows the ellipsoidal or asymmetric distribution of points and the clusters generated by k-means. It's clear that k-means isn't powerful enough to figure out the distribution of these points.

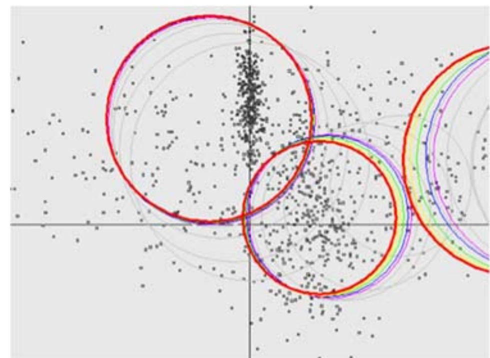


Figure 9.12 Running k-means clustering over an asymmetric normal distribution of points. The points are scattered in an oval area instead of a circular one. K-means can't generate a perfect fit for the data.

Another issue with k-means is that it requires you to estimate *k*, the number of seed centroids, and this value is usually overestimated. Finding the optimal value of *k* isn't easy unless you have a clear understanding of the data, which happens rarely. Even by doing canopy generation, you need to tune the distance measure to make the algorithm improve the estimate of *k*.

What if there was a better way to find the number of clusters? That's where model-based clustering proves to be useful.

ISSUES WITH CLUSTERING REAL-WORLD DATA

Suppose we want to cluster a population of people based on their movie preferences to find like-minded people. We could estimate the number of clusters in such a population by counting the different genres of movies.

Some of the clusters we'd find are people who like action movies, people who like romantic movies, people who like comedies, and so on. This isn't a good estimation, though, because there are tons of exceptions. For example, there are clusters of people who like only gangster movies, not other action movies. They form a subcluster within the action cluster. With such a complex mixing of clusters, we'll never get the information about a small cluster because a bigger cluster would always subsume it. The only way to improve this situation is to somehow understand that a population's movie preferences are hierarchical in nature.

If we had known this earlier, we'd have used a hierarchical clustering method to better cluster the people. But those methods can't capture the overlap. None of the clustering algorithms we've seen so far can capture the hierarchy and the overlap at the same time. What method can we use to uncover this information?

That's another thing that's tackled by model-based clustering.

9.4.2 Dirichlet clustering

Mahout has a model-based clustering algorithm: Dirichlet clustering. The name *Dirichlet* refers to a family of probability distributions defined by a German mathematician, Johann Peter Gustav Lejeune Dirichlet. Dirichlet clustering performs something known as *mixture modeling* using calculations based on the Dirichlet distribution.

The whole process might sound complicated without a deeper understanding of Dirichlet distributions, but the idea is simple. Say you know that your data points are concentrated in an area like a circle and they're well distributed within it, and you also have a model that explains this behavior. You can test whether your data fits the model by reading through the vectors and calculating the probability of the model being a fit to the data.

This approach can say with some degree of confidence that the concentration of points looks more like a circular model. It can also say that the region looks less like a triangle, another model, due to lesser probability of the data matching the triangle. If you find a fit, you know the structure of your data. Figure 9.13 illustrates this concept. (Note that circles and triangles are used here just for visualizing this algorithm. They aren't to be mistaken for the real probabilistic models on which this algorithm works.)

Dirichlet clustering is implemented as a Bayesian clustering algorithm in Mahout. That means the algorithm doesn't just want to give one explanation of the data; rather, it wants to give lots of explanations. This is like saying, "Region A is like a circle, region B is like a triangle, together regions A and B are like a polygon," and so on. In reality, these regions are statistical distributions, like the normal distribution that you saw earlier in the chapter.

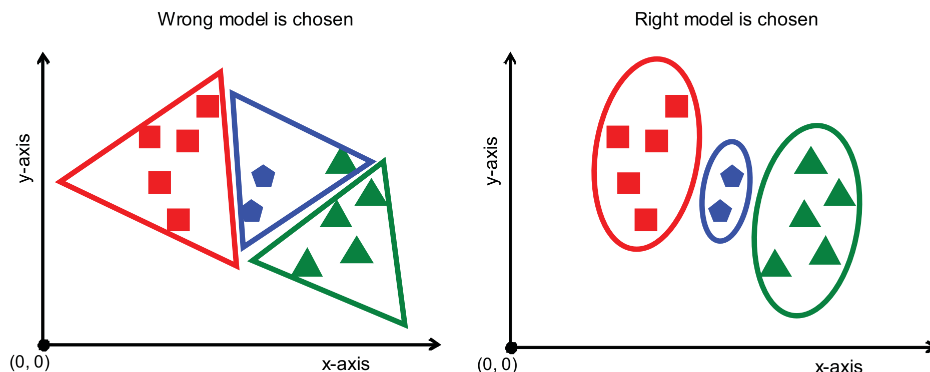


Figure 9.13 Dirichlet clustering: the models are made to fit the given data set as best they can to describe it. The right model will fit the data better and indicates the number of clusters in the data set that correlates with the model.

We'll look at various model distributions a little later in this section, but a full discourse on them is beyond the scope of this book. First, though, let's look at how the Dirichlet clustering implementation in Mahout works.

UNDERSTANDING THE DIRICHLET CLUSTERING ALGORITHM

Dirichlet clustering starts with a data set of points and a `ModelDistribution`. Think of `ModelDistribution` as a class that generates different models. You create an empty model and try to assign points to it. When this happens, the model crudely grows or shrinks its parameters to try and fit the data. Once it does this for all points, it re-estimates the parameters of the model precisely using all the points and a partial probability of the point belonging to the model.

At the end of each pass, you get a number of samples that contain the probabilities, models, and assignment of points to models. These samples could be regarded as clusters, and they provide information about the models and their parameters, such as their shape and size. Moreover, by examining the number of models in each sample that have some points assigned to them, you can get information about how many models (clusters) the data supports. Also, by examining how often two points are assigned to the same model, you can get an approximate measure of how likely these points are to be explained by the same model. Such soft-membership information is a side product of using model-based clustering. Dirichlet clustering is able to capture the partial probabilities of points belonging to various models.

9.4.3 Running a model-based clustering example

Dirichlet process-based clustering is implemented in memory in the `DirichletClusterer` class and as a MapReduce job in `DirichletDriver`. We're going to use the `generateSamples` function (used earlier in section 9.1.2) to create random vectors.

The Dirichlet clustering implementation is generic enough to use with any type of distribution and any data type. The `Model` implementations in Mahout use the `VectorWritable` type, so we'll be using that as the default type in our clustering code.

We'll run the Dirichlet clustering using the following parameters:

- The input `Vector` data is in `List<VectorWritable>` format.
- The `GaussianClusterDistribution` is the model distribution we'll try to fit our data to.
- The alpha value of the Dirichlet distribution is 1.0.
- The number of models (`numModels`) we'll start with is 10.
- The `thin` and `burn` intervals are 2 and 2.

The points will be scattered around a specified center point like a normal distribution, as shown in the following listing.

Listing 9.8 Dirichlet clustering using normal distribution

```
List<VectorWritable> sampleData = new ArrayList<VectorWritable>();

generateSamples(sampleData, 400, 1, 1, 3);
generateSamples(sampleData, 300, 1, 0, 0.5);
generateSamples(sampleData, 300, 0, 2, 0.1);

DirichletClusterer dc =
  new DirichletClusterer(
    sampleData,
      new GaussianClusterDistribution(
        new VectorWritable(new DenseVector(2))),
      1.0, 10, 2, 2);
List<Cluster[]> result = dc.cluster(20);
```

Generate three sets of points

Run Dirichlet clusterer

In listing 9.8, we generate some sample points using a normal distribution and try to fit the normal (Gaussian) model distribution over the data. The parameters of the algorithm determine the speed and quality of convergence.

Here, alpha is a smoothing parameter. A higher value slows down the speed of convergence of the model, so the clustering will have many iterations and will try to overfit the models (the model will find a fitting distribution of data in a smaller area, even if the larger area is the model we may be trying to find). A lower value causes the clustering to merge models more quickly and hence tries to under-fit the model (to create models from points that could ideally have been from two different distributions).

The `thin` and the `burn` intervals are used to decrease the memory usage of the clustering. The `burn` parameter specifies how many iterations to complete before saving the first set of models for the data set. The `thin` parameter decides how many iterations to skip between saving such a model configuration. Many iterations are needed to reach convergence, and the initial states aren't worth exploring. During the initial stages, the number of models is extremely high and they provide no real value, so we can skip them (with `thin` and `burn`) to save memory.

The final state achieved using the `DisplayDirichlet` clustering example is shown in figure 9.14. This class can be found in Mahout's examples folder along with

examples of plenty of other model distributions and their clustering.

The kind of clusters we got here are different from the output of the k-means clustering in section 9.2. Dirichlet clustering did something that k-means couldn't do, which was to identify the three clusters exactly the way we generated them. Any other algorithm would have just tried to cluster things into overlapping or hierarchical groups.

This is just the tip of the iceberg. To show off the awesome power of model-based clustering, we repeat this example with something more difficult than a normal distribution.

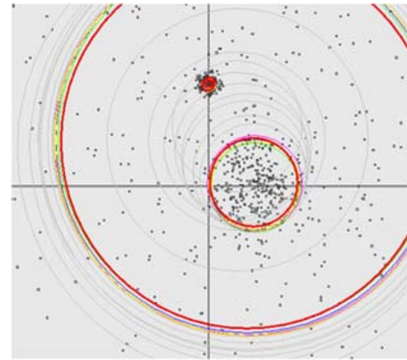


Figure 9.14 Dirichlet clustering with a normal distribution using the `DisplayDirichlet` class in the Mahout examples folder

ASYMMETRIC NORMAL DISTRIBUTION

A normal distribution is asymmetrical when the standard deviations of points along different dimensions are different. This gives it an ellipsoidal shape. When you ran k-means clustering on this distribution in section 9.4.1, it broke down miserably. Now you'll attempt to cluster the same set of points using Dirichlet clustering with `GaussianClusterDistribution`, by keeping the parameters different for x- and y-axes.

You'll run Dirichlet clustering using the asymmetric normal model on the set of 2-dimensional points that has different standard deviations along the x and y directions. The output of the clustering is shown in figure 9.15.

Even though the number of clusters formed seems to have increased, model-based clustering was able to find the asymmetric model and fit the model to the data much better than any of the other algorithms. A better value of alpha might have improved this.

Next, we experiment with the MapReduce version of Dirichlet clustering.

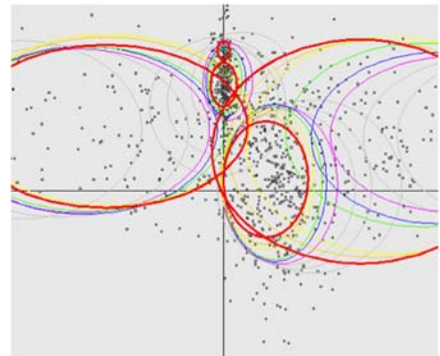


Figure 9.15 Dirichlet clustering with asymmetrical normal distribution illustrated using the `DisplayDirichlet` code in the Mahout examples folder. The thick oval lines denote the final state, and the thinner lines are the states in previous iterations.

MAPREDUCE VERSION OF DIRICHLET CLUSTERING

Like other implementations in Mahout, Dirichlet clustering is focused on scaling with huge data sets. The MapReduce version of Dirichlet clustering is implemented in the `DirichletDriver` class. The Dirichlet job can be run from the command line on the Reuters data set.

Let's get to our checklist for running a Dirichlet clustering MapReduce job:

- The Reuters data set is in `Vector` format.
- The model distribution class (`-md`) defaults to `GaussianClusterDistribution`.
- The type of `Vector` class to be used as the default type for all vectors created in the job (`-mp`). Defaults to `SequentialAccessSparseVector`.
- The `alpha0` value for the distribution should be set to `-a0 1.0`.
- The number of clusters to start the clustering with is `-k 60`.
- The number of iterations to run the algorithm is `-x 10`.

Launch the algorithm over the data set using the Mahout launcher with the `dirichlet` program as follows:

```
$ bin/mahout dirichlet \
-i reuters-vectors/tfidf-vectors \
-o reuters-dirichlet-clusters -k 60 -x 10 -a0 1.0 \
-md org.apache.mahout.clustering.dirichlet.models.GaussianClusterDistribution \
-mp org.apache.mahout.math.SequentialAccessSparseVector
```

After each iteration of Dirichlet clustering, the job writes the state in the output folder in subfolders named with the pattern state-*. You can read them using a `SequenceFile` reader and get the centroid and standard deviation values for each model. Based on this, you can assign vectors to each cluster at the end of clustering.

Dirichlet clustering is a powerful way of getting quality clusters using known data distribution models. In Mahout, the algorithm is a pluggable framework, so different models can be created and tested. As the models become more complex there's a chance of things slowing down on huge data sets, and at this point you'll have to fall back on other clustering algorithms. But after seeing the output of Dirichlet clustering, you can clearly decide whether the algorithm we choose should be fuzzy or rigid, overlapping or hierarchical, whether the distance measure should be Manhattan or cosine, and what the threshold for convergence should be. Dirichlet clustering is both a data-understanding tool and a great data clustering tool.

9.5 Topic modeling using latent Dirichlet allocation (LDA)

So far we've thought of documents as a set of terms with some weights assigned to them. In real life, we think of news articles or any other text documents as a set of topics. These topics are fuzzy in nature. On rare occasions, they're ambiguous. Most of the time when we read a text, we associate it with a set of topics. If someone asks, "What was that news article all about?" we'll naturally say something like, "It talked about the US war against terrorism" instead of listing what words were actually used in the document.

Think of a topic like *dog*. There are plenty of texts on the topic *dog*, each describing various things. The frequently occurring words in such documents might be *dog*, *woof*, *puppy*, *bark*, *bow*, *chase*, *loyal*, and *friend*. Some of these words, like *bow* and *bark* are ambiguous, because they're found in other topics like *bow and arrow* or *bark of a tree*. Still, we can say that all these words are in the topic *dog*, some with more probability than others. Similarly, a topic like *cat* has frequently occurring words like *cat*, *kitten*,

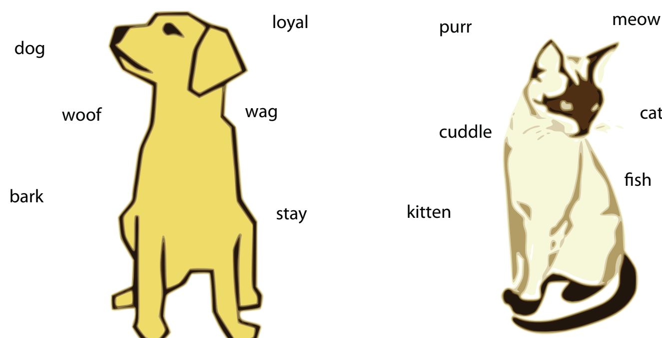


Figure 9.16 The topics *dog* and *cat* and the words that occur frequently in them

meow, *purr*, and *furball*. Figure 9.16 illustrates some words that come to mind when we see a picture of a dog or a cat.

If we were asked to find these topics in a particular set of documents, our natural instinct now would be to use clustering. We'd modify our clustering code to work with word vectors instead of the document vectors we've been using so far. A word vector is nothing but a vector for each word, where the features would be IDs of the other words that occur along with it in the corpus, and the weights would be the number of documents they occur together in.

Once we have such a vector, we could run one of the clustering algorithms, figure out the clusters of words, and call them topics. Though this seems simple, the amount of processing required to create the word vector is quite high. Still, we can cluster words that occur together, call them a topic, and then calculate the probabilities of the word occurring in each topic.

Latent Dirichlet analysis (LDA) is more than just this type of clustering. If two words having the same meaning or form don't occur together, clustering won't be able to associate those two based on other instances. This is where LDA shines. LDA can sift through patterns in the way words occur, and figure out which two have similar meanings or are being used in similar contexts. These groups of words can be thought of as a concept or a topic.

Let's extend this problem. Say we have a set of observations (documents and the words in them). Can we find the hidden groups of features (topics) to explain these observations? LDA clusters features into groups or topics very efficiently.

9.5.1 Understanding latent Dirichlet analysis

LDA is a generative model like Dirichlet clustering. You start with a known model and try to explain the data by refining the parameters to fit the model to the data. LDA does this by assuming that the whole corpus has some k number of topics, and each document is talking about these k topics. Therefore, the document is considered a mixture of topics with different probabilities for each.

NOTE Machine learning algorithms come in two flavors—generative and discriminative. Algorithms like k-means and hierarchical clustering, which try to split the data into *k* groups based on a distance metric, are generally called *discriminative*. An example of the discriminative type is the SVM classifier, which you'll learn about in part 3 of this book. In Dirichlet clustering, the model is tweaked to fit the data, and just using the parameters of the model, you could generate the data on which it fits. Hence, it's called a *generative* model.

LDA is more powerful than standard clustering because it can both cluster words into topics and documents into mixtures of topics. Suppose there's a document about the Olympics, which contains words like *gold*, *medal*, *run*, *sprint*, and there's another document about the 100 m sprints in the Asian games that contains words like *winner*, *gold*, and *sprint*. LDA can infer a model where the first document is considered to be a mix of two topics—one about sports that includes words like *winner*, *gold*, and *medal*, and the other about the 100 m run that includes words like *run* and *sprint*. LDA can find the probability with which each of the topics generates the respective documents. The topics themselves are distributions of the probabilities of words, so the topic *sports* may have the word *run* with a lower probability than in the *100m sprint* topic.

The LDA algorithm works like Dirichlet clustering. It starts with an empty topic model, reads all the documents in a mapper phase in parallel, and calculates the probability of each topic for each word in the document. Once this is done, the counts of these probabilities are sent to the reducer where they're summed, and the whole model is normalized. This process is run repeatedly until the model starts explaining the documents better—when the sum of the (log) probabilities stops changing. The degree of change is set by a convergence threshold parameter, similar to the threshold in k-means clustering. Instead of measuring the relative change in centroid, LDA estimates how well the model fits the data. If the likelihood value doesn't change above this threshold, the iterations stop.

9.5.2 TF-IDF vs. LDA

While clustering documents, we used TF-IDF word weighting to bring out important words within a document. One of the drawbacks of TF-IDF was that it failed to recognize the co-occurrence or correlation between words, such as *Coca Cola*. Moreover, TF-IDF isn't able to bring out subtle and intrinsic relations between words based on their occurrence and distribution. LDA brings out these relations based on the input word frequency, so it's important to give term-frequency vectors as input to the algorithm, and not TF-IDF vectors.

9.5.3 Tuning the parameters of LDA

Before running the LDA implementation in Mahout, you need to understand the two parameters in LDA that impact the runtime and quality.

The first of these is the number of topics. Like the k centroids in k-means, you need to figure out the number of topics based on the data that you have. A lower value

for the number of topics usually results in broader topics like *science*, *sports*, and *politics*, and it engulfs words spanning multiple subtopics. A large number of topics results in focused or niche topics like *quantum physics* and *laws of reflection*.

A large number of topics also means that the algorithm needs lengthier passes to estimate the word distribution for all the topics. This can be a serious slowdown. A good rule of thumb is to choose a value that makes sense for a particular use case. Mahout's LDA implementation is written as a MapReduce job, so it can be run in large Hadoop clusters. You can speed up the algorithm by adding more workers.

The second parameter is the number of words in the corpus, which is also the cardinality of the vectors. This determines the size of the matrices used in the LDA mapper. The mapper constructs a matrix whose size equals the number of topics multiplied by the document length (the number of words or features in the corpus).

If you need to speed up LDA, apart from decreasing the number of topics, you can keep the features to a minimum, but if you need to find the complete probability distribution of all the words over topics you should leave this parameter alone. If you're interested in finding the topic model containing only the keywords from a large corpus, you can prune away the high frequency words in the corpus while creating vectors.

You can lower the value of the maximum-document-frequency percentage parameter (`--maxDFPercent`) in the dictionary-based vectorizer. A value of 70 removes all words that occur in more than 70 percent of the documents.

9.5.4 Case study: finding topics in news documents

We'll run the Mahout LDA over the Reuters data set. First, we run the dictionary vectorizer, create TF vectors, and use them as input for `LDADriver`. The high frequency words are pruned to speed up the calculation. In this example, we'll model 10 topics from the Reuter vectors.

The entry point `LDADriver` takes the following parameters:

- An input directory containing `Vectors`
- An output directory where the LDA states will be written after every iteration
- The number of topics to model, `-k 10`
- The number of features in the corpus, `-v`
- The topic-smoothing parameter, `-a` (uses the default value of 50/number of topics)
- A limit on the maximum number of iterations, `-x 20`

The number of features in the corpus (`-v`) can be easily found by counting the number of entries in the dictionary file located in the vectorizer folder. We can use the `SequenceFileDumper` utility to find the number of dictionary entries, as described in chapter 8. We'll run the LDA algorithm from the command line as follows:

```
$ bin/mahout lda \
-i reuters-vectors/tf-vectors \
-o reuters-lda-sparse \
-k 10 -v 7000 -x 20 -ow
```

LDA will run 20 iterations or stop when the estimation converges. The state of the model after each iteration is written in the output directory in folders whose names begin with state-.

Mahout has an output reader for LDA in the `mahout-utils` module for reading the topic and word probabilities from the output state directory. `LDAPrintTopics` is the main entry point for the utility. You can see the top five words of each topic model from the state folder of any iteration as follows:

```
$ bin/mahout org.apache.mahout.clustering.lda.LDAPrintTopics \
-i reuters-lda-sparse/state-20/ \
-d reuters-vectors/dictionary.file-* \
-dt sequencefile -w 5
```

The output for this example is shown in table 9.1. Note that only 5 topics are shown of the 10.

Table 9.1 Top five words in selected topics from LDA topic modeling of Reuters news data

Topic 0	Topic 1	Topic 2	Topic 3	Topic 4
wheat	south	loans	trading	said
7-apr-1987	said	president	exchange	inc
agriculture	oil	bank	market	its
export	production	chairman	dollar	corp
tonnes	energy	debt	he	company

LDA was able to distill a very diverse set of topics from the Reuters collection. Still, there are some undesired words, like *7-apr-1987*, *said*, *he*, and so on. LDA treats these words like any other words in the collection. More iterations are usually necessary to find better topic models.

The unwanted words don't go away easily because of the high frequency with which they occur. These words belong to any topic with a higher probability than the keywords. This is clear if we try to examine the documents talking about these topics in the corpus. But words like *said* and *he* don't disappear even after pruning high frequency words using the dictionary vectorizer. Can LDA do something better?

One parameter we didn't tweak in this run was the topic-smoothing parameter (`-a`). Because text data is noisy, it induces errors in the LDA estimation. LDA can work around that by increasing the smoothing value to increase the effect of keywords that occur infrequently. Doing this decreases the effect of the high frequency words, as well. This causes LDA to take more iterations to produce a meaningful topic model.

By default, LDA keeps this smoothing parameter set to `50/numTopics`. In our sample run, it was `5`. Let's increase the smoothing value to `20`, and rerun LDA. After the iterations finish, inspect the output using the `LDAPrintTopics` class.

The output is displayed in table 9.2. The effects of high frequency words are still there, but the topics look like they have become more coherent.

Table 9.2 Top five words in selected topics from LDA topic modeling after increased smoothing is applied

Topic 0	Topic 1	Topic 2	Topic 3	Topic 4
production	year	said	stock	vs
tonnes	growth	banks	corp	mln
price	foreign	have	securities	cts
oil	last	analysts	inc	net
department	billion	market	reuter	loss

9.5.5 Applications of topic modeling

Topic modeling output files are of the key-value format (`IntPairWritable`, `DoubleWritable`). The key is a pair of integers, with the first being the topic ID and the second the feature ID; the value is the likelihood of the word being in the model.

We can use these topic models for many practical purposes. We could use them as centroids and associate documents to the nearest center using a distance measure or assign labels to them and use them as models for classification, again using some distance measure.

Topic collections can be visualized as related tag clouds, similar to what Digg and Delicious offer. Topic modeling can also be used to visualize topics across time. We can model topics in news articles by month or by year. We can see trends in topics over time.

> **NOTE** An interesting experiment is the topic modeling of science across time. The most-mentioned words in science journals of the 1890s related to steam engines, in the 1940s they related to atomic research, and in the 1990s they related to polymer and semiconductor devices. This experiment is explained by David M. Blei: http://www.cs.princeton.edu/~blei/topicmodeling.html.

Words in topic models can be used to improve search coverage. For example, a person can search for *Cola*, and get results for the queries *Coca-Cola* and *Pepsi* along with it.

LDA is an algorithm that can uncover interesting clusters and word relationships from a corpus. People are still discovering ways to fully utilize all this information. Mahout LDA helps us analyze millions of documents over a large number of servers. Because it runs very fast, it's easy to experiment with. We explore LDA in a case study in chapter 12 and show how it's used to boost the related document framework.

9.6 Summary

In this chapter you saw the clustering algorithms Mahout had to offer. The various categories of clustering algorithms, based on their clustering strategy, are summarized in table 9.3.

The Mahout implementation of the popular k-means algorithm works well for small and big data sets. A good estimation of the centroids of the clusters makes clustering faster, so we explored ways to improve centroid estimation. The canopy clustering

Table 9.3 The different clustering algorithms in Mahout, their entry-point classes, and their properties

Algorithms	In-memory implementation	MapReduce implementation	Fixed clusters	Partial membership
K-means	`KMeansClusterer`	`KMeansDriver`	Y	N
Canopy	`CanopyClusterer`	`CanopyDriver`	N	N
Fuzzy k-means	`FuzzyKMeansClusterer`	`FuzzyKMeansDriver`	Y	Y
Dirichlet	`DirichletClusterer`	`DirichletDriver`	N	Y
LDA	N/A	`LDADriver`	Y	Y

algorithm did fast and approximate clustering of the data and approximated the centroids of the clusters. By using these centroids as starting points, k-means iterations were found to converge much more quickly than before. We also looked at the various parameters in k-means and used k-means to create a clustering module for a news website. Using the distance measure classes in Mahout we were able to tune the news-clustering module to get better quality clusters for text data.

Fuzzy k-means clustering gives more information about a document's partial membership in various clusters, and it has better convergence properties than k-means. We tuned our clustering module to use fuzzy k-means to help identify this soft membership information. Due to the limitation of fixing a *k* value in both k-means and fuzzy k-means, we also explored other options and found model-based clustering algorithms to be a good replacement for both of them.

The model-based clustering algorithm in Mahout, Dirichlet clustering, didn't just assign points to a set of clusters. It was able to explain how well the model fit the data, as well as the distribution of points in the cluster. This algorithm was able to describe the clusters in some very difficult data sets, where previous methods failed. Dirichlet clustering proved to be a powerful tool to describe such data.

Finally, we looked at LDA, a recent advancement in the area of clustering, which was able to model the data into a mixture of topics. These topics aren't only clusters of documents; they're also probabilistic distributions of words. LDA can jointly cluster the set of words into topics and make the set of documents a mixture of topics. LDA opened up new possibilities, allowing us to identify connections between various words purely from the observed text corpus.

The actual insight into which method works best for your data comes with experimentation. There are powerful tools in the Mahout clustering package, and they're built on top of Hadoop, which gives you the power to scale to cluster data of any size by simply adding more machines.

The next few chapters focus more on tuning a clustering algorithm for speed and quality. Along the way, we refine our news clustering code and finally demonstrate the related-articles feature in action. We also explore some very interesting problems as case studies that the clustering algorithms in Mahout can help solve.

Next, in chapter 10, we look at some lesser known tools and techniques present in Mahout that can help you understand and improve the quality of clustering.

10 Evaluating and improving clustering quality

This chapter covers

- Inspecting clustering output
- Evaluating the quality of clustering
- Improving clustering quality

We saw many types of clustering algorithms in the last chapter: k-means, canopy, fuzzy k-means, Dirichlet, and latent Dirichlet analysis (LDA). They all performed well on certain types of data and sometimes poorly on others. The most natural question that comes to mind after every clustering job is, "How well did the algorithm perform on the data?"

Analyzing the output of clustering is an important exercise. It can be done with simple command-line tools or richer GUI-based visualizations. Once the clusters are visualized and problem areas are identified, these results can be formalized into quality measures, which give numeric values that indicate how good the clusters are. In this chapter, we look at several ways to inspect, evaluate, and improve our clustering algorithms.

Tuning a clustering problem typically involves creating a custom similarity metric and choosing the right algorithm. The evaluation measure shows the impact of changes in the distance measure on clustering quality. First, you need to understand what the cluster looks like and the kind of features that are representative of each centroid. You also need to see the distribution of data-points among different clusters. You don't want to end up in a situation where all k–1 clusters have one point each and the kth cluster has rest of the points.

Once you understand what the clusters look like, you can focus on where you can improve the clustering. This means you need to know how to tune the quality of input vectors, the distance measure, and the algorithm's various parameters.

Let's begin our journey by looking at the tools and techniques that will help you inspect the clustering output.

10.1 Inspecting clustering output

The primary tool in Mahout for inspecting the clustering output is the `ClusterDumper`. It's found in the mahout-utils module in the Mahout codebase under the org.apache.mahout.utils.clustering package (and it's available as the `clusterdump` command in a bin/mahout script). `ClusterDumper` makes it easier to read the output of any clustering algorithm in Mahout. You've already seen it in action in the k-means section of chapter 9, and we'll be using it more in this chapter to understand clustering quality.

The `ClusterDumper` takes the input set of clusters and an optional dictionary that was generated during the conversion of the data to vectors. The full set of options for the `ClusterDumper` tool is shown in table 10.1.

Table 10.1 Flags of the Mahout `ClusterDumper` tool and their default values

Option	Flag	Description	Default value
`SequenceFile` directory (`String`)	`-s`	The directory containing the `SequenceFile` of the clusters.	N/A
Output (`String`)	`-o`	The output file; if not specified, it prints the output to the console.	N/A
Points directory (`String`)	`-p`	At the end of clustering, Mahout clustering algorithms produce two kinds of output. One is the set of <cluster-id, centroid> pairs, and the other is the set of <point-id, cluster-id> pairs. The latter is generated when clustering finishes, and it usually resides in the points folder under the output. When this parameter is set to the points folder, all the points in a cluster are written to the output.	N/A
Dictionary (`String`)	`-d`	The path to the dictionary file that contains the reverse mapping of integer ID to word.	N/A

Table 10.1 Flags of the Mahout `ClusterDumper` tool and their default values *(continued)*

Option	Flag	Description	Default value
Dictionary type (`String`)	`-dt`	The format of the dictionary file. If text, the integer ID and the terms should be tab separated. If the format is `SequenceFile`, it should have an `Integer` key and a `String` value.	text
Number of words (`int`)	`-n`	The number of top terms to print.	10

Recall that any cluster has a centroid point, which is the average of all points in that cluster. This is true for all algorithms, except model-based clustering algorithms like Dirichlet. The center in Dirichlet process clustering isn't really the average of points, but a point that's the anchor for a particular group of data points.

Even then, the centroid points act as summaries of their clusters, and provide quick insight into the nature of each cluster. The top-weighted features of a centroid (its highest-value dimensions) are the features that indicate what that cluster is about. For text documents, the features are words, so the top-weighted words in a centroid vector are words that indicate the meaning of documents in that cluster.

For example, the features from a document talking about *Obama*, *McCain*, and *election* would have higher weight in a cluster of documents about the most recent US presidential election. These top-weighted features are semantically related to each other. On the other hand, if the top-weighted features aren't related, this means the cluster has documented things that do not belong together. This could be caused by many reasons, most of which we'll cover in this chapter.

As an example, let's look at the output of `ClusterDumper` for two outputs of k-means clustering: one using a Euclidean distance measure and the other using a cosine distance measure:

```
$ bin/mahout clusterdump \
-s kmeans-output/clusters-19/ \
-d reuters-vectors/dictionary.file-0 \
-dt sequencefile -n 10
```

`ClusterDumper` outputs a block of information, including the centroid vector and top-weighted terms in the cluster, for each cluster. The dimensions of the vector are translated to the corresponding word in the dictionary and printed on the screen. This output might be more easily examined as a text file, which can be accomplished with the `-o` option:

```
$ bin/mahout clusterdump \
-s kmeans-output/clusters-19/ \
-o output.txt \
-d reuters-vectors/dictionary.file-0 \
-dt sequencefile -n 10
```

The output.txt file can be opened with any text editor. The line containing the centroid vector for each cluster is usually large, so it's omitted in the output below. A typical sorted list of top 10 terms for a cluster generated using k-means clustering over the Reuters collection with a Euclidean distance measure is shown in the next listing.

Listing 10.1 Top 10 terms using k-means clustering with a Euclidean distance measure

```
Top Terms:
                said                          =>   11.60126582278481
                bank                          =>   5.943037974683544
                dollar                        =>    4.89873417721519
                market                        =>   4.405063291139241
                us                            =>  4.2594936708860756
                banks                         =>  3.3164556962025316
                pct                           =>   3.069620253164557
                he                            =>   2.740506329113924
                rates                         =>  2.7151898734177213
                rate                          =>  2.7025316455696204
```

The top terms in the preceding cluster indicate that it contains documents related to banks. A similar cluster is created using a cosine distance measure in the following listing.

Listing 10.2 Top 10 terms using k-means clustering and a cosine distance measure

```
Top Terms:
                bank                          =>  3.3784878432986294
                stg                           =>   3.122450990639185
                bills                         =>   2.440446103514419
                money                         =>   2.324820905806048
                market                        =>   2.223828649332401
                england                       =>  1.6710182027854468
                pct                           =>  1.5883359918481277
                us                            =>   1.490838685054553
                dealers                       =>  1.4752549691633745
                billion                       =>  1.3586127823991738
```

They're similar, but the second cluster appears better than the first one. The second cluster identifies *money* as an important term, whereas the first cluster has a common verb *said* as the top term.

The next section evaluates the quality of clustering by comparing the preceding output.

10.2 Analyzing clustering output

Clustering is a function not only of the input but of several algorithms, each of which has parameters to tune. When clustering goes wrong, there's no single way to debug it. It's important to understand what is going on by analyzing the clusters.

We start by analyzing the output in listings 10.1 and 10.2 by focusing on the following three themes:

- Distance measure and feature selection
- Inter- and intra-cluster distances
- Mixed and overlapping clusters

10.2.1 Distance measure and feature selection

The cosine distance measure is a better distance measure for texts, because it groups documents by the highest-weighted common words between them. The TF-IDF-weighted vectors had higher weights for topic words, so similar documents clustered using the cosine distance measure came to have common topic words between them. This caused the cluster centroid vector to have a higher average weight for topic words than for stop-words.

The top words of listing 10.2 are topical and don't contain stop-words like *said*. We encountered such words in listing 10.1, including some unimportant ones. Though TF-IDF-weighted vectors were used in both instances, the way they were grouped in listing 10.1 gave more importance to the commonly occurring verb *said*. Even though *said* had a small weight, the average weight in the cluster came out to be larger than the topic words because of the equal importance the Euclidean distance measure gives for any feature.

As you can see, choosing a good distance measure helps ensure quality.

WEIGHTING THE FEATURES

Good clusters usually form around a few strong features, which drive a strong notion of similarity amongst the documents. That means choosing the right features is as important as choosing the right distance measure; for unstructured text, better quality is achieved by extracting better features. But there are limitations to what TF-IDF weighting can achieve. For practical applications, we might need to improve upon this weighting technique.

For other types of data, it takes some ingenuity to weight vectors in an optimal way. Good clustering requires that more weight be given to the important features and less weight to the unimportant ones.

Recall the apples to vectors example from chapter 8. Numerically, color had a higher decimal value and weight had a lower value. If we found that weight was a better feature than color, the weights would need to be scaled higher and color lower. For example, the color values could be scaled to a 0–1 range and the weights to a 0–100 range. On doing so, the physical significance of the values vanishes. What remains is a vector with features and weights that will cluster the apples better.

10.2.2 Inter-cluster and intra-cluster distances

Given all centroid points, the distance between all pairs of centroids can be calculated using some distance measure and can be represented in a matrix. This matrix of *inter-cluster distance* gives a nice view of what happened in clustering by showing how near or far the resulting clusters ended up from each other.

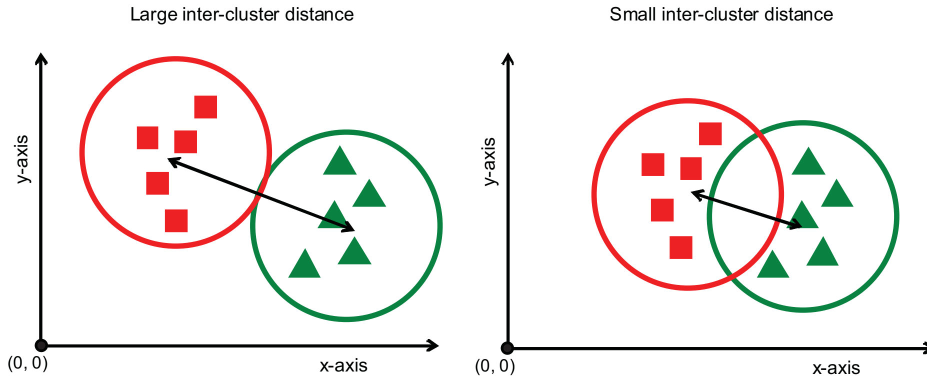


Figure 10.1 Inter-cluster distance is a measure of how well the data is separated. This is dependent on the feature weighting technique and the distance measure used.

Intra-cluster distance is the distance among members of a cluster, rather than the distance between two different clusters. This metric gives a sense of how well the distance measure was able to bring the items together.

Let's look at these two types of distances.

INTER-CLUSTER DISTANCE

Inter-cluster distance is a good measure of clustering quality; good clusters probably don't have centroids that are too close to each other, because this would indicate that the clustering process is creating groups with similar features, and perhaps drawing distinctions between cluster members that are hard to support.

Would it be useful to discover a news article in a cluster on *US*, *president*, and *election*, and find that it doesn't exist in the cluster that talks about *candidate*, *United States*, and *McCain*? Maybe not; we may want clusters that are close to each other. Inter-cluster distance is a metric that correlates with this aspect of quality. Figure 10.1 illustrates the inter-cluster distances across two different configurations of data.

We know that Euclidean distance values are larger, compared to values in the cosine distance measure. Comparing the inter-cluster distances across two distance measures makes no sense unless they're scaled to the same range. The scaled average inter-cluster distances for all cluster centroid pairs are a better measure of clustering quality.

The following listing shows a simple way of calculating the inter-cluster distances from the output of clustering.

Listing 10.3 Calculating inter-cluster distances

```
public class InterClusterDistances {
  public static void main(String args[]) throws Exception {
    String inputFile
        = "reuters-kmeans-clusters/clusters-6/part-r-00000";

    Configuration conf = new Configuration();
    Path path = new Path(inputFile);
```

Set path to clustering output

```
    System.out.println("Input Path: " + path);
    FileSystem fs = FileSystem.get(path.toUri(), conf);

    List<Cluster> clusters = new ArrayList<Cluster>();

    SequenceFile.Reader reader = new SequenceFile.Reader(fs, path, conf);
    Writable key = (Writable) reader.getKeyClass().newInstance();
    Writable value = (Writable) reader.getValueClass().newInstance();

    while (reader.next(key, value)) {                    <- Read cluster object
      Cluster cluster = (Cluster) value;
      clusters.add(cluster);
      value = (Writable) reader.getValueClass().newInstance();
    }

    DistanceMeasure measure = new CosineDistanceMeasure();
    double max = 0;
    double min = Double.MAX_VALUE;
    double sum = 0;
    int count = 0;
    for (int i = 0; i < clusters.size(); i++) {          Calculate distance
      for (int j = i + 1; j < clusters.size(); j++) { <- measures
        double d = measure.distance(clusters.get(i).getCenter(),
            clusters.get(j).getCenter());
        min = Math.min(d, min);
        max = Math.max(d, max);
        sum += d;
        count++;
      }
    }

    System.out.println("Maximum Intercluster Distance: " + max);
    System.out.println("Minimum Intercluster Distance: " + min);
    System.out.println("Average Intercluster Distance(Scaled): "
        + (sum / count - min) / (max - min));
  }
}
```

Listing 10.3 computes the distances between all pairs of centroids and calculates the minimum, maximum, and scaled inter-cluster distances. It produces the following output on the clustering output of `CosineDistanceMeasure` with Reuters:

```
Maximum Intercluster Distance: 0.9577057041381253
Minimum Intercluster Distance: 0.22988591806895065
Average Intercluster Distance(Scaled): 0.7702427093554679
```

When running the same code over the clusters generated using the Euclidean distance measure on the Reuters, the output is markedly different:

```
Maximum Intercluster Distance: 96165.06236154583
Minimum Intercluster Distance: 2.7820472396645335
Average Intercluster Distance(Scaled): 0.09540179823852128
```

Note that the minimum and maximum inter-cluster distances were closer in magnitude when a cosine distance measure was used. This tells us that the clusters are equally spread out when using the cosine distance measure. The minimum inter-cluster distance measure was very small when using the Euclidean distance measure, indicating that at least two clusters are virtually the same or almost completely overlapping.

No. 10

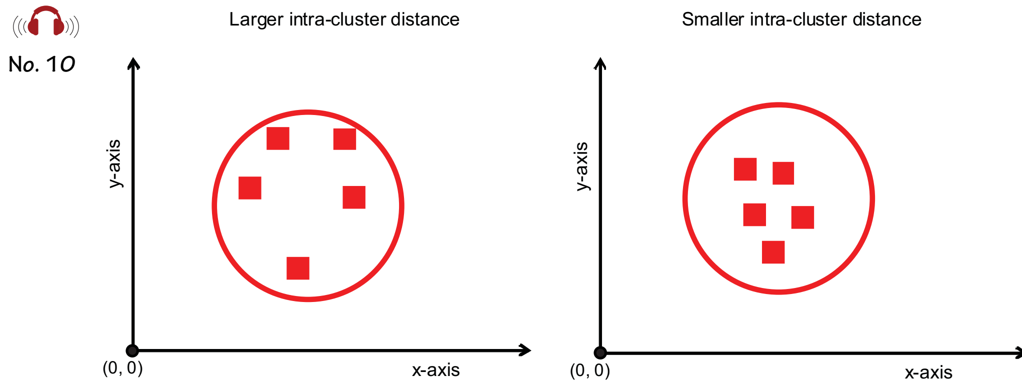


Figure 10.2 **Intra-cluster distance is a measure of how close the points lie to each other. This is dependent on two things: the penalty a distance measure gives for farther objects, and the smaller value it gives for closer objects. The higher the ratio of these two values, the more separated out the clusters are.**

The average scaled inter-cluster distance shows clearly that the cosine distance measure is a better measure of similarity than the Euclidean distance, because it creates clusters that are more evenly separated. This explains why the clusters formed by the cosine distance measure produce top terms of higher quality than when using a Euclidean distance measure.

INTRA-CLUSTER DISTANCE

Intra-cluster distance—the distance between members of a cluster—will be small compared to inter-cluster distances. Figure 10.2 illustrates the intra-cluster distances that might be obtained when clustering using two different distance measures. A good distance measure will return a small distance between objects that are similar and produce clusters that are tighter, and that are therefore more reliably discriminated from one another.

10.2.3 Mixed and overlapping clusters

In some datasets, it's difficult to separate data points into good clusters. The clusters produced in such cases have low inter-cluster distances that are of the same magnitude as the intra-cluster distances. For such datasets, changing the distance measure or improving feature selection usually doesn't make much difference. These datasets require specialized algorithms like fuzzy k-means (to identify partial memberships) or Dirichlet process clustering (to identify a model that fits the data).

It is worth recalling the different clustering algorithms, their strengths, and the types of data they can cluster best. The next section explores some advanced techniques that can improve the inter-cluster and intra-cluster distances.

10.3 Improving clustering quality

There are two important elements in improving the quality of clustering: improving the weights of the features in a document vector and creating a more appropriate distance measure. A good weighting technique can promote the good features of an object, and an appropriate distance measure can help bring similar features together. The next two sections explain how you can create custom feature-selection and distance-measurement classes.

10.3.1 Improving document vector generation

A good document vector has the right kind of features, with higher weights assigned to the more important ones. In text data, there are two ways to improve the quality of a document vector: by removing noise and using a good weighting technique.

Not every text document is of high quality; the sentences could be malformed, or the words might be hard to distinguish because of formatting or structure. This is true for large datasets generated from the web. Most of the textual content on the internet, such as in web pages, blogs, wikis, chats, or discussion forums, is delivered as a complex collection of markup, stylesheets, and scripts, not pure text. Texts generated from OCR (optical character recognition) interpretations of scanned documents and from SMS messages are of very poor quality due to mistaken character recognition (*tne* for *the*) or extreme abbreviations from slangy messaging (*c u* for *see you*). The text might be missing characters, spaces, or punctuation, or it might contain random jargon or even unintended words and grammatical mistakes. Clustering in the presence of such noise is hard. To get optimum quality, the data first needs to be cleaned of these errors.

Some existing text analysis tools do this well. Mahout provides a hook that allows any text filtering technique to be injected into the vectorization process. This is done through something called a Lucene `Analyzer`. You already caught a glimpse of Lucene `Analyzers` in chapter 9 when we tried to improve the news-clustering module, though we didn't explain much of what it did.

Creating a custom Lucene `Analyzer` involves the following:

- Extending the `Analyzer` interface
- Overriding `tokenStream(String field, Reader r)`

The following listing shows a custom Lucene `Analyzer` that can tokenize a document using the `StandardTokenizer`, a somewhat error-resilient tokenizer implementation in Lucene.

Listing 10.4 A custom Lucene `Analyzer` that wraps around `StandardTokenizer`

```
public class MyAnalyzer extends Analyzer {                <- Extend Analyzer

  @Override
   public TokenStream tokenStream(
       String fieldName, Reader reader) {                 <- Make TokenStream from Reader
     TokenStream result = new StandardTokenizer(
```

```
        Version.LUCENE_CURRENT, reader);
    return result;
  }
}
```

The custom analyzer tokenizes the text from the input `Reader` into a stream of tokens. A token is simply a word, so you can think of a text document as a stream of tokens. In listing 10.4 we're using the `StandardTokenizer` to tokenize the contents on the `Reader`.

Next we try improving this `Tokenizer`. Instead of simply tokenizing words, we also add a filter that converts the token into lowercase. The updated class is shown in the following listing.

Listing 10.5 `MyAnalyzer` with lowercase filter

```
public class MyAnalyzer extends Analyzer {

  @Override
  public TokenStream tokenStream(String fieldName, Reader reader) {
    TokenStream result = new StandardTokenizer(
        Version.LUCENE_CURRENT, reader);
    result = new LowerCaseFilter(result);              <- Apply lowercase filter
    return result;
  }
}
```

`TokenFilter` applies a transformation to the underlying token stream, and these filters can be chained. In addition to `LowerCaseFilter`, Lucene offers `StopFilter`, `LengthFilter`, and `PorterStemFilter`. `StopFilter` skips over stop-words in the underlying stream, `LengthFilter` filters out tokens whose lengths are beyond the specified range, and `PorterStemFilter` performs stemming on the underlying word. Stemming is a process of reducing a word into its base form. For example, the words *kicked* and *kicking* are both reduced to *kick*.

> **NOTE** `PorterStemFilter` works only for English texts, but Lucene contains stem filters for other languages too.

Using these filters, a custom Lucene `Analyzer` can be created, as shown in the following listing.

Listing 10.6 A custom Lucene `Analyzer` using multiple filters

```
public class MyAnalyzer extends Analyzer {

  @Override
  public TokenStream tokenStream(String fieldName, Reader reader) {
    TokenStream result = new StandardTokenizer(
       Version.LUCENE_CURRENT, reader);
    result = new LowerCaseFilter(result);           <- Apply lowercase filter
    result = new LengthFilter(result,3, 50);        <- Prune words based on length
    result = new StopFilter(result,
        StandardAnalyzer.STOP_WORDS_SET);           <- Remove stop-words
```

```
      result = new PorterStemFilter(result);       <-- Stem words to base form
      return result;
    }
}
```

Using this Lucene `Analyzer`, we can create TF-IDF vectors and cluster the data from the Reuters collections. We borrow the code from listing 9.4 and use it with this `Analyzer` in the following listing.

Listing 10.7 Modifying NewsKMeansClustering.java to use `MyAnalyzer`

```
public class NewsKMeansClustering {
  public static void main(String args[]) throws Exception {
    int minSupport = 5;
    int minDf = 5;
    int maxDFPercent = 99;
    int maxNGramSize = 1;
    int minLLRValue = 50;
    int reduceTasks = 1;
    int chunkSize = 200;
    int norm = -1;
    boolean sequentialAccessOutput = true;

    String inputDir = "reuters-seqfiles";
    File inputDirFile = new File(inputDir);
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(conf);
    String outputDir = "newsClusters";
    HadoopUtil.delete(conf, new Path(outputDir));
    Path tokenizedPath = new Path(outputDir,
         DocumentProcessor.TOKENIZED_DOCUMENT_OUTPUT_FOLDER);
    MyAnalyzer analyzer = new MyAnalyzer();          <-- Initialize MyAnalyzer
    DocumentProcessor.tokenizeDocuments(
       new Path(inputDir), analyzer.getClass()
      .asSubclass(Analyzer.class), tokenizedPath, conf);   <-- Tokenize using MyAnalyzer

    DictionaryVectorizer.createTermFrequencyVectors(
        tokenizedPath,                                <-- Create TF vectors
        new Path(outputDir), conf, minSupport, maxNGramSize,
        minLLRValue, 2, true, reduceTasks,
        chunkSize, sequentialAccessOutput, false);
    TFIDFConverter.processTfIdf(                      <-- Create IDF vectors
        new Path(outputDir,
            DictionaryVectorizer.DOCUMENT_VECTOR_OUTPUT_FOLDER),
        new Path(outputDir) , conf, chunkSize,
        minDf, maxDFPercent, norm, true, sequentialAccessOutput,
        false, reduceTasks);

    Path vectorsFolder = new Path(outputDir,

                            "/vectors");
    Path centroids = new Path(outputDir,  "/centroids");
    Path clusterOutput = new Path(outputDir,  "/clusters");

    RandomSeedGenerator.buildRandom(conf, vectorsFolder, centroids,
      20, new CosineDistanceMeasure());
```

```
    KmeansDriver.runJob(conf, vectorsFolder, centroids,
      clusterOutput, new CosineDistanceMeasure(), 0.01,
      20, true, false);
```
K-means with random centroids

```
    SequenceFile.Reader reader = new SequenceFile.Reader(fs,
        new Path(clusterOutput, Cluster.CLUSTERED_POINTS_DIR
            + "/points/part-m-00000"), conf);

  }
}
```

This code is similar to code you saw in chapters 8 and 9. Instead of the `StandardAnalyzer`, we used `MyAnalyzer` here and ran k-means clustering over the generated vectors. After running listing 10.7's clustering code on the Reuters data, the top 10 terms in the bank cluster look like this:

```
Top Terms:
                billion                 =>   4.766493374867332
                bank                    =>  2.2411748607854296
                stg                     =>  1.8598368697035639
                money                   =>  1.7500053112049054
                mln                     =>  1.7042239766168474
                bill                    =>  1.6742077991552187
                dlr                     =>  1.5460346601253139
                from                    =>  1.5302046415514483
                pct                     =>  1.5265302869149873
                surplus                 =>  1.3873321058744208
```

Note that the words were stemmed using the `PorterStemFilter` and the `StopwordsFilter` has ensured that the stop-words are virtually absent. This illustrates the power of selecting the right set of features for improving clustering.

10.3.2 Writing a custom distance measure

If the vectors are of the highest quality, the biggest improvement in cluster quality comes from the choice of an appropriate distance measure. We've seen that the cosine distance is a good distance measure for clustering text documents. To illustrate the power of a custom distance measure, we create a different form of the cosine distance measure that exaggerates distances: it makes big distances bigger and small distances smaller. Like writing an `Analyzer`, writing a custom `DistanceMeasure` involves

- Implementing the `DistanceMeasure` interface
- Writing the measure in the `distance(Vector v1, Vector v2)` method

Our modified cosine distance measure is shown in the next listing.

Listing 10.8 A modified cosine distance measure

```
public class MyDistanceMeasure
    implements DistanceMeasure {
  @Override
  public double distance(Vector v1, Vector v2) {
    if (v1.size() != v2.size()) {
      throw new CardinalityException(v1.size(), v2.size());
```
Implement DistanceMeasure interface

Override distance method

```
    }
    double lengthSquaredv1 = v1.getLengthSquared();
    double lengthSquaredv2 = v2.getLengthSquared();

    double dotProduct = v2.dot(v1);

    double denominator = Math.sqrt(lengthSquaredv1)
                          * Math.sqrt(lengthSquaredv2);

    if (denominator < dotProduct) {
      denominator = dotProduct;
    }
    double distance = 1.0 - dotProduct / denominator;     <- Compute distance
    if (distance < 0.5) {
        return (1 - distance) * (distance * distance)
            + distance * Math.sqrt(distance);
    } else return Math.sqrt(distance);                    <- Return distance measure
  }

  @Override
  public double distance(double centroidLengthSquare,
                          Vector centroid,
                          Vector v) {
    return distance(centroid, v);
  }

  @Override
  public void createParameters(String prefix, JobConf jobConf) {}

  @Override
  public Collection<Parameter<?>> getParameters() {
    return Collections.emptyList();
  }

  @Override
  public void configure(JobConf arg0) {}
}
```

The `configure()`, `getParameters()`, and `createParameters()` methods are used to pass arguments to the `DistanceMeasure` at each job node in a MapReduce job. The method of most interest to us is the distance function. Note how it calculates the standard cosine distance measure and then tries to bring closer distances even closer by squaring the cosine distance value (if the value is between 0 and 0.5) and increases the distance otherwise.

Some interesting niche topics surfaced due to this distance measure, which were not there earlier:

```
Top Terms:
                futures            =>  3.2517230513480757
                trading            =>    2.73124880187049
                exchange           =>  2.2752173132458906
                market             =>  2.0752484701513274
                said               =>  1.9316076421017707
                stock              =>  1.8062251904561821
                new                =>  1.5571645992558176
                traders            =>   1.531255338250137
```

```
index                =>  1.3630116680911748
options              =>   1.183384693735014
```

Note that this output was generated from vectors tokenized using `StandardAnalyzer`; you can experiment with both a custom Lucene `Analyzer` and a custom `DistanceMeasure` to get the desired clustering.

10.4 Summary

In this chapter, we explored how clustering can be improved. The algorithms, the distance measures, and the type of data all influence the output of clustering. The `ClusterDumper` tool can inspect output from all the clustering algorithms: the top features of each cluster and the centroid vector. This can help you understand what has happened in the clustering.

For large data sets, it's not feasible to manually inspect all the clusters. Instead, inter-cluster and intra-cluster distances can provide a quick quantitative score of the clustering quality. If many clusters are too close, it's time to move beyond k-means and see how partial membership and mixture distributions can be clustered out.

Writing a custom Lucene `Analyzer` and a custom similarity metric can help improve the clustering quality beyond what the standard implementations can do. We saw that it's worthwhile to explore custom measures to improve clustering.

If you've read this far, you're ready to take on any type of data and tune your clustering algorithm to work well. You'll soon encounter the biggest obstacle in clustering: scale. Fortunately, Mahout can crunch terabytes of data over large Hadoop clusters. Next, we explore how the clustering algorithms in Mahout leverage the power of Hadoop. In chapter 11, you'll see how to run a clustering job over a Hadoop cluster and take your clustering to production.

11 Taking clustering to production

This chapter covers

- Running a clustering job on a Hadoop cluster
- Tuning a clustering job for performance
- Batch clustering versus online clustering

You've seen how different clustering algorithms in Mahout group the documents in the Reuters news data set. Along the way, you learned about the vector representation of data, distance measures, and various other ways to improve the quality of clusters. One of Mahout's strengths is its ability to scale. The Reuters data set wasn't much of a challenge, so in this chapter we set a bigger challenge for Mahout: clustering one of the largest free data sets in the world: Wikipedia—the free encyclopedia. Mahout can handle such scales because the algorithms in it are implemented as MapReduce jobs that can execute on a Hadoop cluster over hundreds and thousands of computers.[1]

[1] Ajay Anand describes running a 4,000 node Hadoop installation at Yahoo! in 2008 in his blog entry, "Scaling Hadoop to 4000 nodes at Yahoo!," http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop _to_4000_nodes_a.html

Unfortunately, not everyone has access to such a cluster. For demonstration purposes in this chapter, we use a subset of documents extracted from Wikipedia and experiment on a small cluster to show how greater speed is achieved by adding more computers. We start by running clustering on a single-machine Hadoop setup, known as a pseudo-distributed setup (which you saw earlier in chapter 6). We also look at the Mahout launcher program, which can easily launch clustering jobs locally or on any Hadoop cluster by providing the configuration file. Then we discuss how a clustering job can be tuned for performance. Finally, we discuss how to use the existing Mahout clustering algorithms and design a system that can incrementally cluster in an online fashion.

11.1 Quick-start tutorial for running clustering on Hadoop

Let's first look at an aspect of Hadoop's architecture.

A Hadoop cluster consists of a single server called the NameNode that handles the different computers in the cluster, which are called DataNodes. The NameNode also synchronizes the Hadoop distributed filesystem (HDFS). Another server called the JobTracker manages all the MapReduce jobs and manages the computers (nodes) where the `Mapper` and `Reducer` classes execute on the cluster. On each of these nodes, another process called the TaskTracker manages the `Mapper` or `Reducer` execution requests from the JobTracker.

Hadoop does all this seamlessly and without any user intervention. In single-node clusters, the NameNode, JobTracker, DataNode, and TaskTracker all run on the same system as separate processes talking with each other.

The version of Hadoop that Mahout is designed to work with at the moment is 0.21 (though it should be compatible with other recent versions).

11.1.1 Running clustering on a local Hadoop cluster

You'll find a short tutorial on setting up a local Hadoop cluster in pseudo-distributed mode on this web page: http://hadoop.apache.org/common/docs/r0.20.2/quickstart.html. You'll need to set up this local cluster for this chapter's examples.

The Hadoop binary resides in the bin/ folder of the Hadoop home directory. To view the contents of the HDFS, execute this command:

```
bin/hadoop dfs -ls /
```

This will list all files in the root of the filesystem.

When the cluster is started for the first time, your home directory might not exist in HDFS, so you should create a /user/<your UNIX user name> folder:

```
bin/hadoop dfs -mkdir /user/<your UNIX username>
```

The home directory can also be listed as follows:

```
bin/hadoop dfs -ls
```

To begin clustering the Reuters data on your pseudo-distributed cluster, prepare the Reuters `SequenceFile` containing the text data locally, as in chapter 8, and copy it to the HDFS:

```
bin/hadoop dfs -put <path-to>/reuters-seqfiles reuters-seqfiles
```

Using this `SequenceFile` as the input, you'll run the dictionary-based vectorizer on the cluster and then run k-means clustering on it.

Any Hadoop MapReduce job is executed using the `hadoop jar` command. Mahout puts all example class files and dependencies in a single JAR file under examples/target/mahout-examples-0.4-SNAPSHOT.job. To run the dictionary vectorizer in the local Hadoop cluster using the Reuters `SequenceFiles` as input, simply execute the Hadoop `jar` command with the Mahout job file as follows:

```
bin/hadoop jar mahout-examples-0.5-job.jar \
  org.apache.mahout.vectorizer.SparseVectorsFromSequenceFiles \
  -ow -i reuters-seqfiles -o reuters-vectors
```

That's it. The clustering executes on the local Hadoop cluster. If the default Hadoop configuration is used, and if the system has at least two processor cores, two mappers will be executed in parallel, with each running on one core. This can be tracked on the JobTracker dashboard at http://localhost:50030, as shown in figure 11.1.

lappy Hadoop Map/Reduce Administration

State: RUNNING
Started: Sun May 02 14:29:32 IST 2010
Version: 0.20.2-dev, r
Compiled: Mon Feb 8 03:33:04 IST 2010 by robinanil
Identifier: 201005021429

Cluster Summary (Heap Size is 26.19 MB/1.74 GB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
0	1	7	1	2	2	4.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete
			mahout-examples-0.4-	100.00%			0.00%

Figure 11.1 A screenshot showing a typical Hadoop MapReduce JobTracker page. You can view this page for your local Hadoop cluster at http://localhost:50030.

11.1.2 Customizing Hadoop configurations

If you have two cores, Hadoop, out of the box, executes clustering twice as fast as the local runs, which we invoked earlier in chapters 8–10. If more than two cores are available on the computer, the Hadoop configuration file can be modified to set the number of mapper and reducer tasks to a higher value, increasing the overall speed of computation. The `mapred.map.tasks` and `mapred.reduce.tasks` configuration properties in the Hadoop installation's mapred-site.xml file needs to be configured to the appropriate value for the number of CPU cores.

With each additional core, the parallelism increases and processing time decreases. Once the number of parallel tasks hits the peak for a single node, increasing the number of tasks decreases performance heavily. The only way to scale more is to have multiple nodes with the same configuration: a fully distributed Hadoop cluster.

> **TIP** You'll find a tutorial with step-by-step instructions for setting up a distributed Hadoop cluster on the Hadoop website: http://hadoop.apache.org/common/docs/r0.20.2/cluster_setup.html. In the 0.20.2 and later versions of Hadoop, the configuration is split across three files in the conf folder: core-site.xml, hdfs-site.xml, and mapred-site.xml. The default configuration values are set in core-default.xml, hdfs-default.xml, and mapred-default.xml. A parameter given in the default file can be overridden in the *-site.xml file. Tuning the MapReduce parameters usually involves editing the mapred-site.xml.

Executing clustering code on a distributed Hadoop cluster is no different from executing it on a single-node cluster. The bin/hadoop script can launch the job on the cluster from the NameNode, slave nodes, or any computer that has access to the NameNode. The only requirement is that the script, when run, needs to access the correct configuration files of the cluster through the `HADOOP_CONF_DIR` environment variable.

EXECUTING JOBS ON A HADOOP CLUSTER USING THE MAHOUT LAUNCHER

Mahout also provides a bin/mahout script (much like the Hadoop launcher script) to launch clustering jobs. We used it extensively in previous chapters to launch clustering as a single-process job. By setting the `HADOOP_HOME` and `HADOOP_CONF_DIR` environment variables, the same script can be used to launch any of the Mahout algorithms on a Hadoop cluster. The script will automatically read the Hadoop cluster configuration files and launch the Mahout job on the cluster.

Typical output will look like this:

```
export HADOOP_HOME=~/hadoop/
export HADOOP_CONF_DIR=$HADOOP_HOME/conf
bin/mahout kmeans -h

running on hadoop, using HADOOP_HOME=/Users/username/hadoop and
     HADOOP_CONF_DIR=/Users/username/hadoop/conf
...
```

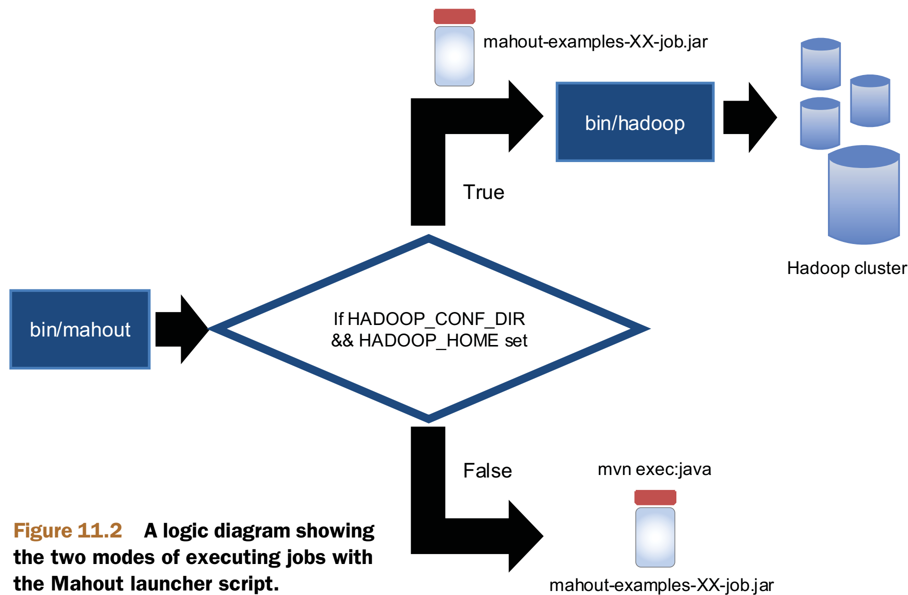


Figure 11.2 A logic diagram showing the two modes of executing jobs with the Mahout launcher script.

The Mahout launcher script internally calls the Hadoop launcher script using the correct cluster configuration files, as is outlined in figure 11.2. In Mahout, the launcher script is the easiest way to launch any algorithm locally or on a distributed Hadoop cluster.

Congratulations, you have successfully run a clustering job on a Hadoop setup. If you're interested, you can spend some time experimenting with various parameters to tune clustering quality. Next, we'll look at how you can tune this setup and achieve higher throughput and performance in clustering.

11.2 Tuning clustering performance

Clustering algorithms in Mahout are designed to run in parallel. Though the algorithms differ, they're similar in one respect: they all read from the `SequenceFile` of vectors in parallel in each `Mapper`.

Many operations in the clustering algorithms are CPU-bound. This means that for many operations, the vector serializations and deserializations, distance computations, and so on, keep the CPU at full usage. On the other hand, some operations are I/O bound, such as transmitting the centroid to each `Reducer` over the network. To improve clustering performance, you need to understand the tricks to address both these types of performance bottlenecks. You'll take a look at how various parameters in Mahout create CPU, disk, or network bottlenecks based on the type of input data.

For reference, the k-means clustering algorithm is illustrated in figure 11.3. Other clustering algorithms like fuzzy k-means, Dirichlet, and LDA all have an architecture similar to k-means, and the optimizations for k-means will hold true for all of them.

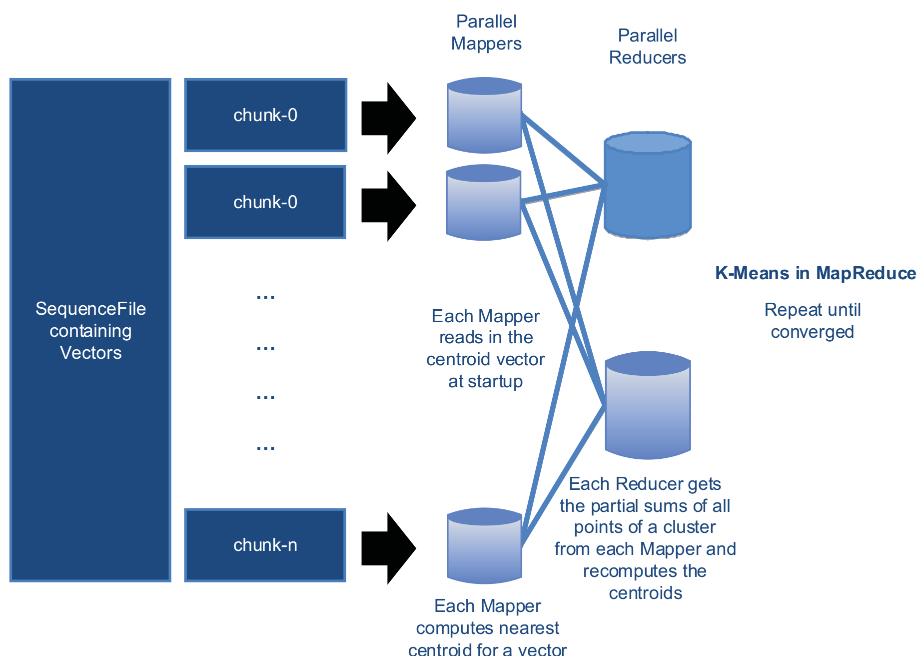


Figure 11.3 Schematic diagram of a k-means clustering iteration running as a MapReduce job

Clustering performance is a function of the input data. To tune performance, you need to analyze various distributions of input and determine the performance pitfalls you might encounter while using certain clustering parameters. Let's begin by looking at how CPU bottlenecks can be reduced, and in some cases completely avoided.

11.2.1 Avoiding performance pitfalls in CPU-bound operations

CPU performance drops when heavily used functions start becoming slower. In clustering, the distance computation is CPU bound, so making the computation faster makes the whole job faster. The following things need to be kept in mind when tackling CPU-related performance problems in clustering.

USING AN APPROPRIATE VECTOR REPRESENTATION

`DenseVector` is generally the fastest `Vector` among the three in Mahout. Any element can be accessed quickly, and it can be iterated efficiently in sequence. But using `DenseVector` for sparse vectors can cause significant performance problems. When used to represent a vector that has many zero elements, a `DenseVector` will wastefully store a lot of elements that would otherwise have no representation in a sparse representation. This means more data is needlessly serialized, and many zero elements are iterated over in operations that wish to ignore zero elements.

For example, say you're clustering very sparse data (the number of non-zero elements is about 1 percent of the cardinality) like text document vectors. The clustering using `SparseVector` is about twice as fast as `DenseVector` on the local system and

about 10 times faster on distributed systems. The extra performance during distributed clustering is due to the latency caused by transferring unnecessary 0 bytes over the network in a `DenseVector`.

As a result, it's usually best to represent data as a `SparseVector` because even with smaller sparseness, it offers great savings in storage, deserialization, and network operations.

USING A FAST DISTANCE MEASURE

Clustering algorithms calculate distances often, so using a fast distance-measure implementation is crucial. Improvements in the distance measure's speed go straight to the bottom line of the algorithm's overall performance.

If you implement your own distance measure, follow these best practices:

- Avoid cloning or instantiating a new `Vector`. `Vector`s are heavy Java objects, and cloning them degrades performance heavily.
- Avoid iterating on all elements if the distance measure requires only non-zero elements. Use the `Vector.iterateNonZero()` iterator instead of `Vector.iterator()`.
- Use `Vector.assign()` or `Vector.aggregate()` methods to efficiently traverse and modify the vector. You can learn more about them in appendix B.

USING A SPARSEVECTOR TYPE BASED ON THE DISTANCE-MEASURE COMPUTATION

There are two sparse vector implementations, and it pays to use the implementation appropriate to the distance-measure computation. `RandomAccessSparseVector` is tuned for random lookups, and `SequentialAccessSparseVector` is tuned for fast sequential access.

For example, computing a cosine distance measure entails many vector dot product operations, which requires iterating over two vectors in order, element by element. The code needs to multiply values from matching indices from both vectors. Naturally, `SequentialAccessSparseVector` is optimal for such a sequential access pattern in the distance measure, and it's much faster than `RandomAccessSparseVector`. Keeping all the document vectors in sequential format gives a huge boost to the speed of distance-measure computation and hence improves the overall clustering performance.

> **TIP** The Mahout utils package has a utility class called `VectorBenchmarks`. It pits different types of vector operations across combinations of dense, random-access and sequential-access vectors and compares their speed. If your custom distance measure does more of certain types of vector operations, you can use this utility to find the vector that performs that operation best at a certain sparsity level. The tool is found in the mahout-utils module in the org.apache.mahout.benchmark package.

11.2.2 Avoiding performance pitfalls in I/O-bound operations

I/O performance is usually affected most by the amount of data the program reads and writes. In the case of Hadoop jobs, these reads and writes are mostly sequential.

Decreasing the number of bytes written improves overall clustering performance greatly. The following things need to be kept in mind to decrease the I/O bottleneck.

USING AN APPROPRIATE VECTOR REPRESENTATION

This is quite obvious. As explained earlier, you should never store sparse vectors as `DenseVector` objects. It bloats the size on disk and can create both disk and network I/O bottlenecks for the clustering application.

USING HDFS REPLICATION

Any file stored on HDFS is by default replicated three times across various nodes of the cluster. This is done to guard against data loss, in case a copy on one node is lost, and it also increases performance. If data exists on only one node, all `Mappers` and `Reducers` that need that data must access that one node. This can be a bottleneck. Replication allows HDFS to store the chunks on separate servers, giving Hadoop a choice of nodes where the computation could be initialized, thereby reducing this network I/O bottleneck. Replication can be increased to potentially further reduce this bottleneck, but that will mean using more storage on HDFS. Consider this only if there is evidence that under-replication is creating a bottleneck.

REDUCING THE NUMBER OF CLUSTERS

Clusters are usually represented as large, dense vectors, each of which consumes considerable storage. If the clustering job is trying to find a large number of clusters (k), these vectors are sent across the network from the `Mappers` to the `Reducers`. Having a small value of k will decrease this network I/O. It also decreases the overhead of the distance-measure computation in k-means and fuzzy k-means, where each cluster centroid increases the distance-measure computation by the number of points stored on the disk.

When it's necessary to cluster data into large numbers of clusters, you can throw it all at Hadoop, add more computers, and let it scale for itself. But you can smartly reduce the distance-measure computation and clustering time by an order of magnitude by using a two-tiered approach for batch clustering. This can help speed up clustering for very large data sets, like Wikipedia.

Alternatively, we can explore more incremental clustering methods to reduce the amount of data that needs to be examined for clustering. This can help online aggregators, like news websites, to automatically add articles to a cluster as and when they arrive. These two approaches are explained in the next section.

11.3 Batch and online clustering

Let's return to the AllMyNews.com online news portal case study from chapter 9. We found that a related-articles feature implementation could be constructed as a simple clustering problem. But clustering won't give us the desired final output. Assume the portal has about a million news articles. We'd have to produce clusters containing up to, say, 100 articles, or else the list of related articles will become unusably large. But that implies producing 10,000 clusters.

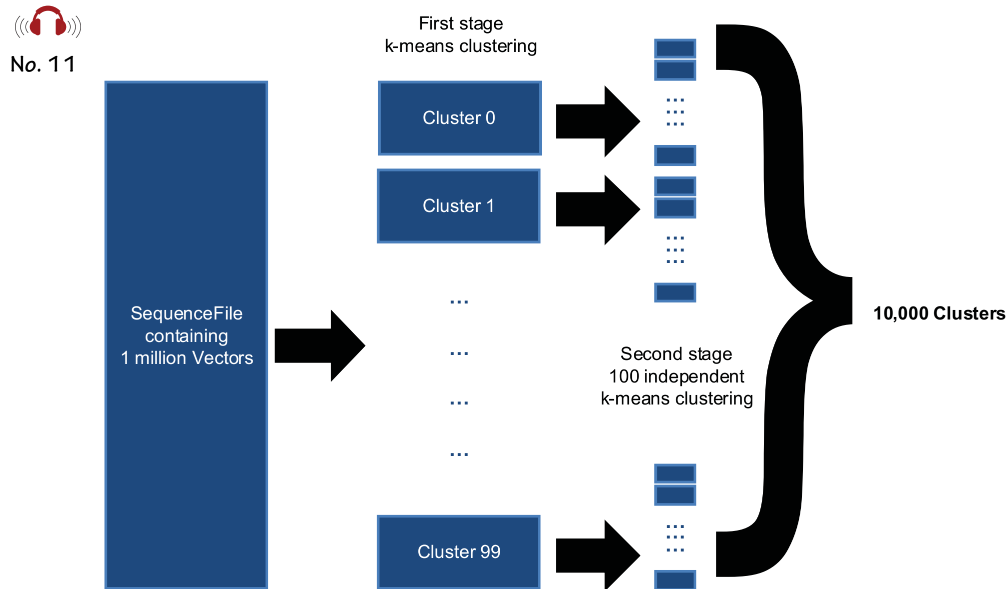


Figure 11.4 Applying k-means in two levels. A single large MapReduce with 10,000 clusters will have $Nx10^6x10^4$ distance computations, where *N* is the number of iterations. On the other hand, the hierarchical breakdown of the problem gives rise to only $N_1x10^6x10^2$ + $N_2x10^4x10^2x10^2$ distance computations—almost 100 times fewer computations.

This isn't a good way to go about solving this problem. Although Hadoop enables us to scale up to do this, it would consume a wasteful amount of CPU and disk, which cost money. A better way to go about this would be to split the articles into 100 giant clusters. Then, for each cluster of about 10,000 articles, cluster again to get 100 smaller clusters, as illustrated in figure 11.4.

As in past examples, we're clustering all articles in one big effort. This is known as batch clustering, or offline clustering. But as new articles come into the news portal, they need to be clustered too. It would be quite expensive to keep running the batch clustering jobs again and again just to cluster a few recent articles. Even if it were run hourly, we'd be unable to identify articles related to a new article for up to an hour. This is clearly not desirable behavior.

Enter *online clustering techniques*, which aim to efficiently cluster new items right away, given a clustering of existing items.

11.3.1 Case study: online news clustering

Online clustering here isn't truly online or truly immediate. There are algorithms that directly cluster streams of data, but they have trouble scaling. Instead, consider the following technique:

- Cluster one million articles, as discussed previously, and save the cluster centroids for all clusters.

- Periodically, for each new article, use canopy clustering to assign it to the cluster whose centroid is closest, based on a very small distance threshold. This ensures that articles on topics that occurred previously are associated with that topic cluster and are shown instantly on the website. These documents are removed from the new document list.
- The leftover documents that aren't associated with any old clusters form new canopies. These canopies represent new topics that have appeared in the news and that have little or no match with any articles from the past.
- Use the new canopy centroids, cluster the articles that aren't associated with any of the old clusters, and add these temporary cluster centroids to the centroid list.
- Less frequently, execute the full batch clustering to recluster the entire set of documents. While doing so, use all previous cluster centroids as input to the algorithm so that clustering achieves faster convergence.

The key here is to ensure that the incremental canopy clustering is done as quickly as possible, so that the user gets related clusters within minutes of a story being published. One dedicated node can easily take care of this. Alternatively, you could delay publishing the story on the website by a couple of minutes until the related-article clusters are generated.

This example shows that just because a solution can use up lots of resources easily and can scale doesn't mean it should use lots of resources. We could have run full clustering on a dedicated cluster with thousands of nodes, but this would have just wasted money. A cheaper and faster solution is available with a little ingenuity.

11.3.2 Case study: clustering Wikipedia articles

Wikipedia's articles are exported every night into XML files and made available here: http://dumps.wikimedia.org/enwiki/. This is an amazing data set. Since it's being edited and organized by humans, many articles remain insufficiently or incorrectly grouped within Wikipedia. You'll try to run clustering on these articles to extract the related articles, which could potentially help the editors group the related articles on the Wikipedia website.

Before attempting to cluster the articles, you need to extract the documents and vectors from the XML format. Fortunately, Mahout has a Wikipedia data set creator class that can read the XML format and output each individual article in `SequenceFile` format. The extractor can also select documents matching a certain category. For the purposes of this experiment, you'll extract all documents in the science category.

First, download the latest pages-articles.xml.bz2 file from the previously mentioned website. Extract it to a local directory, and upload it to the HDFS. Assuming the Hadoop environment variables are set, run the Wikipedia extractor as follows:

```
echo "science" > categories.txt
bin/mahout seqwiki -c categories.txt -i articles.xml \
  -o wikipedia-seqfiles -e
```

This will select all documents from the article XML files that have a Wikipedia category exactly matching the word *science* and write those articles to a `SequenceFile`. You can run the `SparseVectorsFromSequenceFiles` job over this output to create the Wikipedia TF-IDF vectors. Then you can run any clustering algorithm on top of it. This whole process will take quite a while.

Once you have successfully run the clustering, you may wish to experiment with the clustering algorithm in order to observe the effect of, say, the choice of vector representation. Recreate the vectors as `SequentialAccessSparseVector` with the `-seq` flag in the dictionary vectorizer. This will run faster than the `RandomAccessSparseVector`, which is created as default.

Once you're confident enough in clustering the Wikipedia sample set, extract the full data set and vectorize it as follows:

```
bin/mahout seqwiki -all -i articles.xml -o wikipedia-seqfiles
```

Unless you have access to a powerful Hadoop cluster, this may take up to a whole a day on one machine. Another option is to run it in the cloud, as you'll see next.

RUNNING WIKIPEDIA CLUSTERING ON AMAZON ELASTIC MAPREDUCE

One cheap way to explore Mahout is to purchase Hadoop cluster time from providers in the cloud like Amazon's Elastic MapReduce. Section 6.6 of this book showed you a quick way to access this resource.

If you're comfortable with the Amazon Elastic MapReduce (EMR) service, Mahout algorithms can be run as follows:

1. Upload the Mahout job file (mahout-examples-0.5-job.jar) to an Amazon S3 bucket.
2. Provide the path to the *-job.jar file as the JAR file.
3. Specify the full class name for the driver class of the algorithm that needs to be run—perhaps `org.apache.mahout.clustering.kmeans.KMeansDriver`—and the parameters.
4. Specify the input path of the Wikipedia `Vectors` file stored on an S3 bucket and the output path on the S3 bucket where clusters need to be written.
5. Specify the other required parameters for the MapReduce, like the number of machines. Remember, each compute unit is charged by the hour, so a large cluster quickly gets expensive.
6. Run the job.

You can follow the same procedure to generate vectors directly using the EMR service by uploading the sequence file and running the `SparseVectorsFromSequenceFile` class.

At the time of writing, it took an eight-node cluster about an hour to convert Wikipedia `SequenceFiles` into vectors. The clustering time depends heavily on the initial centroids, the distance measure, and the number of clusters—and, of course, the number of machines available to work on the computation. Always remember, the computation isn't free.

11.4 Summary

In this chapter, we took a brief look at a Hadoop cluster and how a clustering job is executed on it. We also looked at the Mahout launcher script and its two modes of operation.

We discussed the various issues in clustering performance and analyzed various techniques to bring down the CPU and I/O clustering bottlenecks. Correct vector representation and optimized distance measurement are the keys to high performance clustering applications.

We also applied this knowledge to complete the pseudo-online clustering engine for our fictitious news aggregation portal—AllMyNews.com. With a bit of ingenuity, the large clustering problem was broken down into simpler and faster ones using a mix of techniques, thus achieving near-real-time clustering of related articles in the system. To highlight Mahout's ability to scale clustering, we tried to cluster the largest public database available today, the English Wikipedia. Because the process was slow, we took a quick look at how multi-node clustering could be performed in the cloud using the Amazon Elastic MapReduce service.

With this, you've learned to run Mahout on a Hadoop cluster and make it scale. The journey in this part of the book wasn't simple. We started in chapter 7 with a basic clustering of points in a plane, from which you learned about vectors and distance measures. We toured various algorithms in Mahout and tried to apply them to a real-world case study, which gradually evolved from small input and non-distributed computation to large-scale distributed computation. You also learned techniques for tuning the algorithms for speed and quality. Next, in chapter 12, you'll use all you've learned and apply it to real-world case studies.

12 Real-world applications of clustering

This chapter covers

- Clustering like-minded people on Twitter
- Suggesting tags for an artist on Last.fm using clustering
- Creating a related-posts feature for a website

You probably picked up this book to learn and understand how clustering can be applied to real-world problems. So far we've mostly focused on clustering the Reuter's news data set, which had around 20,000 documents, each having about 1,000 to 2,000 words. The size of that data set isn't challenging enough for Mahout to show its ability to scale. In this chapter, we use clustering to solve three interesting problems on much larger data sets.

First, we attempt to use the public tweets from Twitter (http://twitter.com) to find people who tweet alike using clustering. Second, we examine a data set from Last.fm (http://last.fm), a popular Internet radio website, and try to generate related tags from the data. Finally, we take the full data dump of a popular technology discussion website, Stack Overflow (http://stackoverflow.com), which has

around 500,000 questions and 200,000 users. We use this data set to implement related-features functionality for the website.

Our first problem is finding similar users by clustering tweets from Twitter.

12.1 Finding similar users on Twitter

Twitter is a social networking and micro-blogging service where people broadcast short, public messages known as tweets. These tweets are at most 140 characters long. Once posted, a tweet is instantly published on the feed of all of the user's followers and it's visible publicly on Twitter's site. These tweets are sometimes annotated with keywords in a style like `#Obama` or marked as directed at another user with syntax like `@SeanOwen`.

Because Twitter's new terms of service don't allow us to publish data sets, you'll need to prepare it yourself. In the book's source code, there's a class called `Twitter-Downloader` that grabs 100,000 tweets from Twitter (though you can change this number in the source code) and writes them to a file that later will be used for analysis. Fetching 100,000 tweets will take about 2 hours.

> **NOTE** To run this program, you'll need to obtain an authentication key from Twitter by registering your application at https://dev.twitter.com/apps, and putting the authentication values into the program, as described in the source code. And don't forget to add the `-Dfile.encoding=UTF-8` option when you run this program—otherwise you'll lose all non-ASCII characters.

12.1.1 Data preprocessing and feature weighting

This data must be preprocessed to put it into a format usable by Mahout. First it must be converted to the `SequenceFile` format that can be read by the dictionary vectorizer. Then this must be converted into vectors using TF-IDF as the feature-weighting method.

The dictionary vectorizer expects its input to have a `Text` key and value. Here, the key will be the Twitter username, and the value will be the concatenation of all tweets from that user.

Preparing this input is a job well suited to MapReduce. A `Mapper` reads each line of the input: username, timestamp, and the tweet text separated by tabs. It outputs the username as the key and the tweet text as the value. A `Reducer` receives all tweets from a particular user, joins them into one string, and outputs this as the value, with the username again as the key. This data is written into a `SequenceFile`.

The `Mapper`, the `Reducer`, and the `Configuration` classes, along with a `Job` that invokes them on a Hadoop cluster, are shown in listings 12.1 and 12.2. The MapReduce classes are implemented as a generic group-by-field `Mapper`.

Listing 12.1 Group-by-field mapper

```
public class ByKeyMapper extends Mapper<LongWritable,Text,Text,Text> {
  private Pattern splitter = Pattern.compile("\t");
  private int selectedField = 1; // tweet
  private int groupByField = 0; // username

  @Override
  protected void map(LongWritable key, Text value,
 Context context) throws IOException,
InterruptedException {
    String[] fields = splitter.split(value.toString());      // Split line using Regex
    if (fields.length - 1 < selectedField ||
        fields.length - 1 < groupByField) {
      context.getCounter(
          "Map", "LinesWithErrors").increment(1);              // Count line errors
      return;
    }

    String oKey = fields[groupByField];
    String oValue = fields[selectedField];
    context.write(new Text(oKey), new Text(oValue));          // Output username, tweet

  }

}
```

Listing 12.2 Group-by-field reducer

```
public class ByKeyReducer extends Reducer<Text,Text,Text,Text> {
  @Override
  protected void reduce(Text key,
                      Iterable<Text> values,
                      Context context) {
    StringBuilder output = new StringBuilder();
    for (Text value : values) {                                 // Read user's tweet
      output.append(value.toString()).append(" ");              // Generate concatenated string
    }
    context.write(key, new Text(
        output.toString().trim()));                              // Output concatenated string
  }
}
```

Once the job executes, the user's tweets are concatenated into a document-sized string and written out to the SequenceFile. Next, this output must be converted to TF-IDF vectors. Before running the dictionary-based vectorizer, it's worth understanding the Twitter data set better and identifying some simple and helpful tweaks that will aid the feature-selection and weighting process.

12.1.2 Avoiding common pitfalls in feature selection

TF-IDF is an effective feature-weighting technique that we've used several times already. If it's applied to individual tweets, most term frequencies would be 1 because the text is 140 characters in length, or perhaps 25 total words. We're interested in

clustering users that tweet alike, so we've instead created a data set where a document consists of all the tweets for a user, instead of individual tweets. Applying TF-IDF to this will make more sense, and the clustering will happen more meaningfully.

Directly applying TF-IDF weighting may not produce great results. It would be better to remove features that commonly occur in all tweets: stop-words. We use the dictionary vectorizer and set the maximum document frequency percentage parameter to a low value of perhaps 50 percent. This removes words that occur in more than 50 percent of all documents. We also use the collocations feature of the vectorizer to generate bigrams from the Twitter data and use them as features for clustering.

But there is a problem. Tweets are casually written messages, not carefully edited text like that in the Reuters news collection. Tweets contain misspellings, abbreviations, slang, and other variations. Three people tweeting the sentences below will never get clustered together because their tweets have no words in common:

```
HappyDad7: "Just had a refreshing bottle of Loke"
BabyBoy2010: "I love Loca Kola"
Cool_Dude9: "Dude me misssing LLokkkeee!!!!"
```

To solve this problem, we can discover the relationship between these alternate spellings using phonetic filters. These are algorithms that reduce a word to a base form that approximates the way the word is pronounced. For example, *Loke* and *Loca* might both be reduced to something like *LK*. The word *refreshing* could be reduced to something like *RFRX*. There are at least three notable implementations of a procedure like this: Soundex, Metaphone, and Double Metaphone. All are implemented in the Apache Commons Codec library.[1] We use its `DoubleMetaphone` implementation to convert each word into a base phonetic form. The following listing shows a sample Lucene `Analyzer` implementation using `DoubleMetaphone` in this way.

Listing 12.3 Lucence `Analyzer` class optimized for Tweets

```
public class TwitterAnalyzer extends Analyzer {
  private DoubleMetaphone filter = new DoubleMetaphone();      <-- Initialize encoder
  @Override
  public TokenStream tokenStream(String fieldName, Reader reader) {
    final TokenStream result = new PorterStemFilter(new StopFilter(
        true, new StandardTokenizer(Version.LUCENE_CURRENT, reader),
        StandardAnalyzer.STOP_WORDS_SET));

    TermAttribute termAtt = (TermAttribute) result
        .addAttribute(TermAttribute.class);
    StringBuilder buf = new StringBuilder();
    try {
      while (result.incrementToken()) {
        String word = new String(termAtt.termBuffer(), 0, termAtt
            .termLength());
        buf.append(filter.encode(word)).append(" ");      <-- Create phonetic root
```

[1] You can find the hierarchy for the Apache Commons Codec package at http://commons.apache.org/codec/apidocs/org/apache/commons/codec/language/package-tree.html

```
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
    return new WhitespaceTokenizer(new StringReader(buf.toString()));
  }
}
```

> **NOTE** The `DoubleMetaphone` filter will only work for English text. You might need to find a filter if you deal with other languages. When you'll grab tweets from Twitter, the collection will also contain tweets in other languages. We ignore the quality loss due to other languages for this example.

Copy the `SequenceFile` generated in section 12.1.1 to the cluster using the Hadoop HDFS copy command. Once the data is copied to the cluster, you can run the `DictionaryVectorizer` on the Twitter `SequenceFiles` as follows:

```
bin/mahout seq2sparse -ng 2 -ml 20 -s 3 -x 50 -ow \
-i mia/twitter_seqfiles/ -o mia/twitter-vectors -n 2 \
-a mia.clustering.ch12.TwitterAnalyzer -seq
```

It's important to make sure the `TwitterAnalyzer` class is in the Java `CLASSPATH` when you try to run this command. Vectorization may take some time to tokenize the tweets and generate the complete set of bigrams from them. The vectorized representation will be smaller than the original input.

We don't know the number of clusters we need to generate. Because fewer than 100,000 unique users generated those tweets, we can decide to create clusters of 100 users each, and generate around 1,000 clusters. Run the k-means algorithm using the vectors you just generated. An example of a cluster from the output we got when we ran this is listed below:

```
Top Terms:
        KMK                      =>   7.609467840194702
        APKM                     =>  7.5810538053512575
        TP                       =>   6.775630271434784
        KMPN                     =>   5.795418488979339
        MRFL                     =>  4.9811420202255245
        KSTS                     =>   4.082704496383667
        NFLX                     =>   4.005469512939453
        PTL                      =>  3.9200561702251435
        N                        =>  3.7717770755290987
        A                        =>  3.4771755397319795
```

Here the feature *KMK* is the output of the `DoubleMetaphone` filter for words like *comic*, *comics*, *komic*, and so on. To see the original words, run the clustering with vectors without using any of the filters, or simply using the `StandardAnalyzer`.

You can rerun clustering twice more with the maximum document frequency set at 50 percent and at 20 percent. When we did so, the following top-terms lists were selected from the cluster containing tweets about the topic *comics*:

```
Top Terms with -maxDFPercent=50
        comic                   =>   9.793121272867376
        comics                  =>   6.115341078151356
        con                     =>   5.015090566692931
        http                    =>   4.613376768249454
        i                       =>   4.255820257194115
        sdcc                    =>   3.927590843402978
        you                     =>   3.635386448917967
        rt                      =>  3.0831371437419546
        my                      =>  2.9564724853544524
        webcomics               =>   2.916910980686997

Top Terms with -maxDFPercent=20
        webcomics               =>  11.411304909842356
        comic                   =>  10.651778670719692
        comics                  =>  10.252182320186071
        webcomic                =>   7.813077817644392
        con                     =>   5.867863954816546
        http                    =>   4.937416185651506
        companymancomic         =>   4.899269049508231
        spudcomics              =>   4.543261228288923
        rt                      =>   4.149479137148176
        addanaccity             =>   4.056342724391392
```

These two sets of output illustrate the importance of feature selection in a noisy data set like tweets. With the maximum document frequency set at 50 percent of the document count, there are features like *i*, *you*, *my*, *http*, *rt*. These features dominate the distance-measure calculations, so they appear with higher weights in the cluster centroid. Decreasing the document-frequency threshold removed many of these words. Still, there are words like *http* and *rt* that appear in the second cluster, which should ideally be removed. But setting a much lower threshold runs the risk of removing important features. The best option is usually to manually create a list of words and prune them away using the Lucene `StopFilter`: words like *bit.ly*, *http*, *tinyurl.com*, *rt*, and so on.

This is clustering using only words to calculate user similarity. But similarity can also be inferred from user interaction. For example, if user *A* retweeted (forwarded, or repeated) user *B*'s tweet, you might count *B* as a feature in *A*'s user vector, with a weight equal to the number of times *A* retweeted *B*'s tweets. This may be a good feature for clustering users who think alike.

Alternatively, if the feature weight were the number of times the users mentioned each other in a tweet, adding this as a feature might negatively affect the clustering result. Such a feature is best for clustering people into closed communities that tweet with each other, and not for clustering users who think alike. The best feature for one problem may not be good for others.

The clustering result will show how various tech bloggers are clustered together and how various friend groups are clustered together. We leave it to you to inspect the results and experiment with the clustering. Next, we take a look at another interesting problem: a tag-suggestion engine for a web 2.0 radio station.

12.2 Suggesting tags for artists on Last.fm

Last.fm is a popular Internet radio site for music. Last.fm has many features; of interest here is a user's ability to tag a song or an artist with a word. For example, the band Green Day could be tagged with *punk*, *rock*, or *greenday*. The band The Corrs might be tagged with words like *violin* and *Celtic*.

We now try to implement a related-tags function using clustering. This could assist users in tagging items; after entering one tag, the system could suggest other tags that seem to go together with that tag in some sense. For example, after tagging a song with *punk*, it might suggest the tag *rock*. We explore how to do this in two ways: first, with a simple co-occurrence–based approach, and then with a form of clustering.

We use as input a data set that records the number of times each artist was tagged with a certain word. The data set is available for download from the web at http://musicmachinery.com/2010/11/10/lastfm-artisttags2007/. It contains raw tag counts for the 100 most frequently occurring tags that Last.fm listeners have applied to over 20,000 artists during that period. The data set has this format:

```
artist-id<sep>artist-name<sep>tag-name<sep>count
```

Each line in the data set contains an artist ID and name, a tag name, and the count of times that tag was applied to the artist.

12.2.1 Tag suggestion using co-occurrence

We can use co-occurrence, or the number of times two tags appear together for some artist, as the basis of a similarity metric between tags. We saw co-occurrence first in chapter 6, where we counted item co-occurrence. With a similarity metric, we can then cluster.

For each tag, we could count co-occurrence with all other tags and present the most co-occurring tags as suggestions. There is only one problem. A frequently occurring tag like *awesome* will co-occur with many other tags, so it would become a suggestion for many tags, even though it isn't a particularly helpful suggestion. From that perspective, we might have to cut out high-frequency tags.

This technique, however, will never suggest tags that are two or more hops away. That is, if tag *A* co-occurs with *B* and *B* co-occurs with *C*, this method won't suggest tag *C* to the person adding tag *A*. A workaround is to add all the tags that are two hops away from tag *A* to the suggestion list, and rerank the tags.

But clustering takes care of this out of the box. If the tags are within a certain distance threshold, regardless of whether they co-occur together, they're clustered.

We can model this problem as artist-based clustering where the artist is the document and the feature is the tag the artist receives. Alternatively, you can think of tags as documents, the artists as words in the document, and the number of times they co-occur as the word frequency. These are alternative views of the same problem and can be modeled like a bipartite graph as shown in figure 12.1.

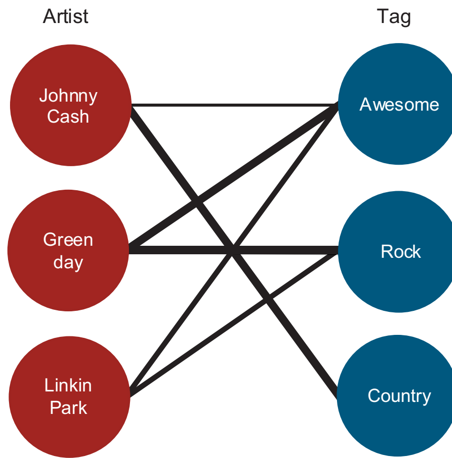


Figure 12.1 A typical bipartite graph (a graph where nodes can be put in one of two partitions). Here, one type of node is for artists and the other is for tags. The edges carry a weight equal to the number of times a tag was applied to the artist. In the world of recommenders, the nodes were users and items, and the edge weights denoted the star ratings. Recommenders and clustering algorithms are more closely related than many people realize.

The Last.fm data set already looks like it can be directly converted to vectors. We create a simple vectorizer that first generates a dictionary of artists, and, using the dictionary, convert the artists into integer dimensions in a MapReduce fashion. We could have used the strings as IDs, but Mahout `Vector` format only takes integer dimensions. The reason for converting artists into integer dimensions will be clear once we arrive at the code that vectorizes Last.fm artists into vectors of their tags.

12.2.2 Creating a dictionary of Last.fm artists

To generate these feature vectors for the Last.fm data set, we employ two MapReduce jobs. The first generates the unique artists in the form of a dictionary, and the second generates vectors from the data using the generated dictionary. The `Mapper` and `Reducer` classes of the dictionary generation code are shown in listings 12.4, 12.5 and 12.6.

Listing 12.4 Output artists from the data set

```
public class DictionaryMapper extends
Mapper<LongWritable,Text,Text, IntWritable> {
  private Pattern splitter;

  @Override
  protected void map(LongWritable key, Text line, Context context)
       throws IOException,
                 InterruptedException {
    String[] fields = splitter.split(line.toString());
    if (fields.length < 4) {
      context.getCounter("Map", "LinesWithErrors").increment(1);
      return;
    }
    String artist = fields[1];                           // Select artist from split line

    context.write(new Text(artist), new IntWritable(0));  // Emit artist as key
  }
```

```
  @Override
  protected void setup(Context context) throws IOException,
                                      InterruptedException {
    super.setup(context);
    splitter = Pattern.compile("<sep>");
  }
}
```

The Mapper takes a line of text and outputs the artist as the key with an empty value. This is passed into the reducer.

Listing 12.5 Grouping by artist and outputting a unique key-value pair for an artist

```
public class DictionaryReducer extends
Reducer<Text,IntWritable,Text,IntWritable> {
  @Override
  protected void reduce(Text artist,
                        Iterable<IntWritable> values,
                        Context context) throws IOException,
                                        InterruptedException {
    context.write(artist, new IntWritable(0));      // Selects only key
  }
}
```

The `Mapper` emits the artist, and the `Reducer` emits only a unique artist and discards the list of empty strings. Once the dictionary is generated, it can be read into a `HashMap`, which maps the words to a unique index as the value, as shown in the following listing. This is done sequentially.

Listing 12.6 Using the unique artist names to create a unique integer ID for each artist

```
  ...
    Map<String,Integer> dictionary = new HashMap<String,Integer>();
    FileSystem fs = FileSystem.get(dictionaryPath.toUri(), conf);
    FileStatus[] outputFiles = fs.globStatus(
new Path(dictionaryPath, "part-*"));            // Select all dictionary files
    int i = 0;
    for (FileStatus fileStatus : outputFiles) {
      Path path = fileStatus.getPath();
      SequenceFile.Reader reader = new SequenceFile.Reader(fs, path,
          conf);
      Text key = new Text();
      Text value = new Text();
      while (reader.next(key, value)) {
        dictionary.put(key.toString(),
            Integer.valueOf(i++));              // Put unique tag in dictionary
      }
    }
  ...
```

This `HashMap` is serialized and stored in the Hadoop configuration. It then goes into the next `Mapper`, which uses this as a dictionary to convert string tags to integer dimension IDs, as you'll see next.

12.2.3 Converting Last.fm tags into Vectors with musicians as features

Using the dictionary generated in the previous section, we create vectors of tags so that they can be clustered together. The vectorization process is a simple MapReduce job. The `Mapper` selects the integer feature ID of an artist and creates a vector with a single feature. These partial vectors with one dimension are sent to the `Reducer`, which simply joins the partial `Vector` into a full `Vector`, as shown in the following listing.

Listing 12.7 Using integer ID mapping of the artist to convert tags into a vector

```
public class VectorMapper extends
    Mapper<LongWritable,Text,Text,VectorWritable> {
  private Pattern splitter;
  private VectorWritable writer;
  private Map<String,Integer> dictionary = new HashMap<String,Integer>();

  @Override
  protected void map(LongWritable key, Text value,
      Context context) throws IOException,
                              InterruptedException {
    String[] fields = splitter.split(value.toString());
    if (fields.length < 4) {
      context.getCounter("Map", "LinesWithErrors").increment(1);
      return;
    }
    String artist = fields[1];
    String tag = fields[2];
    double weight = Double.parseDouble(fields[3]);

    NamedVector vector = new NamedVector(
        new SequentialAccessSparseVector(dictionary.size()), tag);

    vector.set(dictionary.get(artist), weight);
    writer.set(vector);

    context.write(new Text(tag), writer);
  }

  @Override
  protected void setup(
      Context context) throws IOException,
                              InterruptedException {
    super.setup(context);

    Configuration conf = context.getConfiguration();

    DefaultStringifier<Map<String,Integer>> mapStringifier
        = new DefaultStringifier<Map<String,Integer>>(
              conf, GenericsUtil.getClass(dictionary));
    dictionary = mapStringifier.fromString(conf.get("dictionary"));
    splitter = Pattern.compile("<sep>");
     writer = new VectorWritable();
  }
}
```

Annotations: Select fields from split; Create vector; Write ID, weight into vector; Output partial vector; Setup runs before Mapper; Deserialize HashMap

The SequentialAccessSparseVector class is used as the underlying format for representing the tags vector. K-means performs many sequential passes over the vector elements, and this representation is best suited to that access pattern. Using the partial vectors from the Mapper as input, the Reducer simply concatenates them into a full vector, as seen in the following listing.

Listing 12.8 Grouping partial vectors of tags and summing them up into full vectors

```
public class VectorReducer extends
    Reducer<Text,VectorWritable,Text,VectorWritable> {
  private VectorWritable writer = new VectorWritable();

  protected void reduce(Text tag,
                        Iterable<VectorWritable> values,
                        Context context) throws IOException,
                                        InterruptedException {
    Vector vector = null;
    for (VectorWritable partialVector : values) {
      if (vector == null) {
        vector = partialVector.get().like();          // Initialize empty vector
      }
      partialVector.get().addTo(vector);          // Merge partial Vectors to full Vector
    }

    NamedVector namedVector = new NamedVector(vector, tag
        .toString());
    writer.set(namedVector);

    context.write(tag, writer);          // Output full vector
  }
}
```

The VectorReducer merges the partial vectors and converts them into a full Vector. The Last.fm artist vectors are now ready. The next step is to run k-means to generate the topics.

12.2.4 Running k-means over the Last.fm data

The vectorization process generates around 98,999 unique tags from the Last.fm data set. You might want to cluster this into small clusters having about 30 tags each.

To start with, we try to create 2,000 tag clusters from k-means as follows:

```
bin/mahout kmeans -i mia/lastfm_vectors/ \
-o mia/lastfm_topics -c mia/lastfm_centroids -k 2000 -ow \
-dm org.apache.mahout.common.distance.CosineDistanceMeasure \
-cd 0.01 -x 20 -cl
```

On one machine, it may take about 30 minutes to complete the iterations. This is a CPU-bound process, so adding more worker machines in the Hadoop cluster will improve its completion time.

Once they have been output, the tag clusters can be inspected on the command line using the clusterdump utility using the following command:

```
bin/mahout clusterdump -s mia/lastfm_topics/clusters-16 \
-d mia/lastfm_dict/part-r-00000-dict -dt sequencefile -n 10
```

These are top artists for a given tag:

```
Top Terms:
        Shania Twain                  =>   1.126984126984127
        Garth Brooks                  =>   0.746031746031746
        Sara Evans                    =>  0.6031746031746031
        Lonestar                      =>  0.5238095238095238
        SHeDaisy                      => 0.47619047619047616
        Faith Hill                    =>  0.4126984126984127
        Miranda Lambert               => 0.36507936507936506
        Dixie Chicks                  => 0.36507936507936506
        Brad Paisley                  =>  0.3492063492063492
        Lee Ann Womack                =>  0.3492063492063492

Top Terms:
        Explosions in the Sky         =>   55.19047619047619
        Joe Satriani                  =>   51.19047619047619
        Apocalyptica                  =>   50.95238095238095
        Steve Vai                     =>  43.666666666666664
        Mogwai                        =>   39.57142857142857
        Godspeed You! Black Emperor   =>   32.00100000000101
        Yann Tiersen                  =>  30.857142857142858
        Pelican                       =>  20.285714285714285
        Do Make Say Think             =>  16.904761904761905
        Liquid Tension Experiment     =>   16.61904761904762

Top Terms:
        Justin Timberlake             =>   6.739130434782608
        Disturbed                     =>   0.391304347826087
        Timbaland                     => 0.34782608695652173
        Nelly Furtado                 => 0.30434782608695654
        Lustans Lakejer               =>  0.2608695652173913
        Michael Jackson               =>  0.2608695652173913
        Aaliyah                       => 0.21739130434782608
        Timbaland Magoo               => 0.21739130434782608
        Jonathan Davis                => 0.21739130434782608
        Sum 41                        => 0.21739130434782608
```

These tags could be stored in a database and later queried to produce suggestions for any tag.

This example demonstrated the trick of transforming a data set into a different view in order to apply clustering. Our next example is a bit more involved and works on a much larger, open data set of a popular Q & A website: Stack Overflow.

12.3 Analyzing the Stack Overflow data set

Stack Overflow (http://stackoverflow.com) is an online discussion website hosting questions and answers on a wide range of programming-related topics. Users ask and answer questions and vote for or against questions and answers. Users also earn reputation points for interacting on the website. For example, a user gets 10 points upon receiving an up vote on an answer given to a question. After accumulating enough points, users earn additional abilities on the site.

Thanks to Stack Overflow, all data on the website is freely available for noncommercial research. We explore Stack Overflow as a data set, and consider some of the interesting problems that can be modeled as clustering: for example, clustering questions to find related questions, clustering users to find related users, and clustering similar answers to make the answers more readable.

12.3.1 Parsing the Stack Overflow data set

The entire data sets of the stackoverflow.com site and its sister sites (superuser.com, serverflow.com, and so on), known as Stack Exchange, are available as a single file at http://blog.stackoverflow.com/category/cc-wiki-dump. The data sets are around 3.5 GB, compressed. All the data sets are similar in nature, with Stack Overflow being the largest and most prominent of the group. Uncompressed, the Stack Overflow data set is around 5 GB, split across multiple XML files. There are separate files for posts, comments, users, votes, and badges.

Parsing XML files can be tricky, especially in a Hadoop MapReduce job. Mahout provides `org.apache.mahout.classifier.bayes.XMLInputFormat` for reading an XML file in a MapReduce job. It correctly detects each repeated block of XML and passes it to the `Mapper` as an input record. This string will have to be further parsed as XML using any XML parsing library.

The `XMLInputFormat` class needs the starting and ending tokens of the XML block. This is achieved by setting the following keys in the Hadoop `Configuration` object:

```
...
conf.set("xmlinput.start", "<row Id=");
conf.set("xmlinput.end", " />");
...
conf.setInputFormat(XmlInputFormat.class);
...
```

In the `Mapper`, set the key as `LongWritable` and the value as `Text`, and read the entire XML block as `Text` in the value, including the start and end.

12.3.2 Finding clustering problems in Stack Overflow

The Stack Overflow data set contains a table of users, a table of questions, and a table of answers. There are relations between them, based on the user-id, question-id, and answer-id fields that will help you to sift and transform the data in whichever way you'd like. There are some interesting clustering problems for this data.

CLUSTERING POSTS DATA FOR SURFACING RELATED QUESTIONS

It's useful to identify questions that concern similar topics. Users browsing a post that doesn't quite address their question might find a better answer among a list of related questions. This process of finding similar questions will be like finding similar articles in the Reuters data set. These posts are smaller than Reuters articles but larger than Twitter tweets, so TF-IDF weighting ought to give good features. It's always good to remove high-frequency words from the data set, so keep the maximum document frequency threshold at 70 percent or even lower.

One big challenge you'll encounter while vectorizing this data set is the lack of good tokenization of the Stack Overflow questions. Many questions and answers have snippets of code from different programming languages, and the default `StandardAnalyzer` isn't designed to work on such data. You'll have to write a parser that takes care of braces and arrays in code and other quirks in different programming languages.

Instead of using just the question, you can bundle together both questions and their answers, as well as comments, to produce larger documents and get more features to cluster similar questions. Unlike Twitter, the spelling mistakes should not harm the quality very much due to the larger size of the content. But adding a `DoubleMetaphone` filter would give a slight improvement in quality. Because there is lot of data, both k-means and fuzzy k-means should yield similar results. A better improvement in quality will only come by using LDA topics as features, but the CPU consumed by LDA may be prohibitively high for this data set.

CLUSTERING USER DATA FOR SURFACING SIMILAR USERS

Imagine you're a developer who extensively uses the Java Messaging Service (JMS) API. It might be useful to discover people who work with JMS as well. Helping users form such communities not only improves the user experience on the website, but also drives user engagement. Again, clustering can help compute such possible communities.

For clustering users, you need a vector from the features of the user. This could be the kind of content the users posts or answers, or the kind of interaction the user has with other users. These are some of the features of the vector:

- The contents of the questions or answers the user created, which includes the *n*-grams from text and code snippets
- The other users who replied or commented on the post the user created and interacted on.

Clustering users could be done using just the post content, by using just the common interaction counts, or both. Previously, when clustering tweets, only the content was used. It would be a good exercise to experiment on using the interaction features to cluster users.

In this model, a user is a document and its features are the IDs of other users who that user interacted with; the feature value corresponds to how much other users interacted with them. This can be weighted. For example, if user 2 commented on user 1's post, this might map to a weight of 10. But if user 2 voted on user 1's post, it might just be weighted a 2. If the vote was negative, it might receive a weight of –10 to indicate that these two users have a lower probability of wanting to interact with each other. Think of all possible interactions and assign weights to them. The weight of the feature user 2 in the vector of user 1 would be the cumulative weight of the interaction they have had. With these vectors and features, try clustering users based on their interaction patterns and see how it works out.

12.4 Summary

In this chapter, we examined real-world clustering by analyzing three data sets: Twitter, Last.fm, and Stack Overflow.

We started with the analysis of tweets by trying to cluster users who tweet alike. We preprocessed the data, converted it to vectors, and used it to successfully cluster users by their similarity in tweets. Mahout was able to scale easily to a data set of 10 million tweets due to the MapReduce-based implementations.

We took a small subset of a data set based on Last.fm tags and tried to generate a tag-suggestion feature for the site. We modeled the problem as a bipartite network of music artists on one part and user-generated tags on the other, with the edge weights being the number of times they co-occurred together. We clustered based on tags as well. Using this output, it was easy to generate the tag-suggestion functionality.

Finally, we examined data from Stack Overflow. We discussed solving several real-life problems on such a discussion forum using only clustering.

This concludes our coverage of clustering algorithms in Mahout. This was a gentle introduction to one aspect of machine learning, which started with the basic structures and mathematics and progressed to solving large-scale problems on a Hadoop cluster.

Now we'll move on to discuss classification with Mahout, which entails more complex machine learning theory and some solid optimization techniques—both distributed and non-distributed—to achieve state-of-the-art classification accuracy and performance.

Part 3

Classification

This third and final part of *Mahout in Action*, comprising chapters 13 through 17, covers how to use Mahout for classification. Using the techniques presented here, you'll be able to structure questions and choose and prepare data appropriately to have machines automatically assign data to preselected categories. Classification is a simplified form of decision making that gives discrete answers to an individual question. Machine-based classification is an automation of this decision making process that learns from examples of correct decision making and emulates those decisions automatically—a core concept in predictive analytics. Classification's reliance on guided learning and focus on answering one question at a time distinguish it from clustering and recommendation, discussed in the previous two parts of this book. Clustering, in contrast to classification, relies on machines to decide on their own; recommenders select and rank the best of many possible answers.

This part of the book takes you through three stages in classification. Chapters 13 and 14 introduce the basics and show you how to build and train a Mahout classification model. Chapter 15 explains how to use evaluation throughout the process to identify the best versions of your model and fine-tune it for best performance. Chapters 16 and 17 show you how to deploy your classification system in real-world situations. Techniques to correctly identify and extract useful data are presented, along with detailed discussions of how to optimize the features as vectors usable by the various Mahout classification algorithms. Tips for avoiding problems such as target leaks are also included. Step-by-step examples give you experience with each step to help you build classifiers and fine-tune them for best performance.

In addition to the detailed explanation of how to build and deploy an effective and reliable classifier for systems with huge requirements for speed and scale, we present in chapter 17 a real-world case study of an online marketing company that uses Mahout classification to address its needs. This final real-world case study lets you see how the ideas presented throughout the classification chapters can be applied.

13 Introduction to classification

This chapter covers

- Why Mahout is a powerful choice for classification
- Key classification concepts and terminology
- The workflow of a typical classification project
- A step-by-step classification example

Life often presents us with questions that aren't open-ended but instead ask us to choose from a limited number of options. This relatively simple idea forms the basis for classification—both that done by humans and that done by machines. Classification relies on the categorization of potential answers, and machine-based classification is the automation of such simplified decisions.

This chapter identifies those cases where Mahout is a good approach for classification and explains why it offers an advantage over other approaches. As an introduction to classification, this chapter also explains what classification is, and provides a foundation to understanding the basic terminology and concepts. In addition, we present a practical overview of how classification works, covering three stages in the workflow for typical classification projects: training the model,

evaluating and tuning the model, and using the model in production. Following this introduction, we look at a simple classification project and show you, in a simple step-by-step way, how to put these basic ideas into practice.

13.1 Why use Mahout for classification?

Mahout can be used on a wide range of classification projects, but the advantage of Mahout over other approaches becomes striking as the number of training examples gets extremely large. What large means can vary enormously. Up to about 100,000 examples, other classification systems can be efficient and accurate. But generally, as the input exceeds 1 to 10 million training examples, something scalable like Mahout is needed. Table 13.1 shows the situations in which Mahout is most appropriate.

Table 13.1 Mahout is most useful with extremely large or rapidly growing data sets where other solutions are least feasible.

System size in number of examples	Choice of classification approach
< 100,000	Traditional, non-Mahout approaches should work very well. Mahout may even be slower for training.
100,000 to 1 million	Mahout begins to be a good choice. The flexible API may make Mahout a preferred choice, even though there is no performance advantage.
1 million to 10 million	Mahout is an excellent choice in this range.
> 10 million	Mahout excels where others fail.

The reason Mahout has an advantage with larger data sets is that as input data increases, the time or memory requirements for training may not increase linearly in a non-scalable system. A system that slows by a factor of 2 with twice the data may be acceptable, but if 5 times as much data input results in the system taking 100 times as long to run, another solution must be found. This is the sort of situation in which Mahout shines.

In general, the classification algorithms in Mahout require resources that increase no faster than the number of training or test examples, and in most cases the computing resources required can be parallelized. This allows you to trade off the number of computers used against the time the problem takes to solve.

The conceptual graph in figure 13.1 illustrates the advantages of scalable algorithms such as those in Mahout for medium to very large scale classification projects. When the number of training examples is relatively small, traditional data mining approaches work as well or better than Mahout. But as the number of examples increase, Mahout's scalable and parallel algorithm is better with regard to time. The increased time required by non-scalable algorithms is often due to the fact that they require unbounded amounts of memory as the number of training examples grows. Extremely large data sets are becoming increasingly widespread. With the advent of more electronically stored data, the expense of data acquisition can decrease

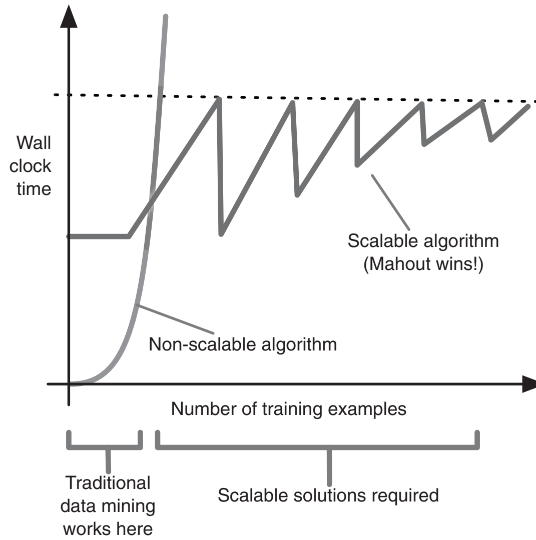


Figure 13.1 The advantages of using Mahout for medium and large classification systems. The jagged line shows the effect of recruiting new machines to increase performance. Each new machine decreases training time.

enormously. Using increased data for training is desirable, because it typically improves accuracy. As a result, the number of large data sets that require scalable learning is increasing, and as it does, the scalable classifiers in Mahout are becoming more and more widely useful.

13.2 The fundamentals of classification systems

Before looking at the way Mahout-based classification is done, let's delve a little deeper into the basics of what classification is, and how it differs from recommendation and clustering.

Let's first consider familiar examples of human-based classification. Filling in medical histories, completing surveys about the quality of a business's service, or checking off the correct box for your filing status on tax forms—these are all situations where you make choices from a limited group of preselected answers. Sometimes the number of categories for responses is very small, even as few as two in the case of yes/no questions. In these situations, the classification process reduces what are potentially large and complicated responses to fit into a small number of options.

> **DEFINITION** Classification is a process of using specific information (input) to choose a single selection (output) from a short list of predetermined potential responses.

Compare what happens to a person at a wine tasting to a person going to the store to get wine for dinner. At the wine tasting, people will ask, "What do you think of that wine?" and a wide range of answers are likely to be heard: "peppery," "overtones of fruit," "a rich finish," "sharp," "the color of my aunt's sofa," and "nice label." The question is nonspecific, and the answers (output) don't form a single selection from a predetermined short list of possibilities. This situation isn't classification, though it may have some similarities to classification.

Now consider the wine shopper at the grocery store. Some of the same wine characteristics discussed at the wine tasting might matter to the shopper (taste, color, and so on), but in order to efficiently make the decision (to buy or not buy) and get to dinner, the shopper most likely has a mental checklist of simple characteristics to look for (red, $10, good with pizza, a familiar label), and for each bottle makes the decision (buy or don't buy). Some detail and nuance may be lost, but simplifying the question to one that has very few potential answers (buy, don't buy) is a practical way to make many decisions manageable. In this case, it gets the person out of the store in time for dinner. This situation is a form of classification.

Machines can't do all the types of thinking and decision-making that humans do. But when the situation is within specific boundaries, machines can emulate human decisions efficiently in order to carry out classification. This machine-based approach can work when there is a single, focused decision, the input to be considered consists of a specific set of characteristics, and the output is categorical, which is to say, a single selection from a short list.

Classification algorithms are at the heart of what is called *predictive analytics*. The goal of predictive analytics is to build automated systems that can make decisions to replicate human judgment. Classification algorithms are a fundamental tool for meeting that goal.

One example of predictive analytics is spam detection. A computer uses the details of user history and features of email messages to determine whether new messages are spam or are relatively welcome email. Another example is credit card fraud detection. A computer uses the recent history of an account and the details of the current transaction to determine whether the transaction is fraudulent.

How do automated systems do this? We'd prefer them to do it magically, but in lieu of that, we rely on having them learn by example.

> **DEFINITION** Computer classification systems are a form of machine learning that use learning algorithms to provide a way for computers to make decisions based on experience and, in the process, emulate certain forms of human decision making.

Classification algorithms learn by example, but they aren't a substitute for human judgment, largely because they require carefully prepared examples of correct decisions from which to learn, and they require inputs that are carefully prepared for use in these algorithms. Unlike some other uses of Mahout, machine-based classification is a form of supervised learning.

13.2.1 Differences between classification, recommendation, and clustering

Classification algorithms stand in contrast to the clustering algorithms described in previous chapters because clustering algorithms are able to decide on their own which distinctions appear to be important (in machine learning parlance, they're unsupervised learning algorithms) whereas classification algorithms learn to mimic examples

of correct decisions (they're supervised learning algorithms). Classification algorithms are also different from recommendation algorithms because they're intended to make a single decision with a very limited set of possible outcomes, whereas recommendation algorithms select and rank the best of many possible alternatives.

> **WARNING** One way to think about what classification *is* is to remind yourself of what classification *is not*: If your system is unsupervised, it *is not* classification. If your questions are open-ended or the answers aren't categorical, your system *is not* classification.

Somewhat surprisingly, classification can be used creatively and effectively as a tool in a recommendation process. Typically, using a classifier to build a recommendation system isn't as efficient as a real recommendation system, but the classifier as recommender can use some kinds of information that a normal recommendation system would have a hard time using. Recommendations systems normally use a history of what users have done, ignoring user and item characteristics that can be useful input for a classification system. The case study in chapter 17 will describe how a classifier can be useful as a recommender.

We've begun to answer the question, "What is classification?" Next we'll look at what classification is used for.

13.2.2 Applications of classification

The output of a classification system is the assignment of data to one of a small predetermined set of categories. The usefulness of classification is usually determined by how meaningful the prechosen categories are. By assigning data to categories, the system allows the computer or a user to act on the output.

Classification is often used for prediction or detection. In the case of credit card fraud, a classifier can be useful in predicting which transactions will later be determined to be fraudulent based on known examples of behavior for purchases and other credit card transactions. The yes/no decision of whether or not a particular transaction is fraudulent is made by assigning that transaction to one of two categories: yes, fraud, or no, not fraud.

Classification is also valuable in predicting customer attrition in industries such as insurance or telecommunication. One way to design a classification system for this application is to train on data from a previous year's records of known cases of attrition or retention, which are the categories being targeted as answers. Known characteristics of customers included in the training data set are selected as the features that can best predict whether or not a customer will stay with the service. Once trained, the classification system can be used to predict the future behavior of other customers based on the known, specific characteristics, including current behavior.

> **NOTE** Keep in mind that the term *prediction*, as used to describe what classifiers do, doesn't refer solely to the ability to foresee future events. It also refers to the estimation of a value that may exist but not be easily available at the time of the classification.

Prediction can be the task of correctly assigning a characteristic (a category) based on available data rather than by directly testing for the characteristic. Often the characteristic that's targeted for classification is itself too expensive to be determined directly, and that's why classification becomes useful as a predictor. By expensive, we mean too costly in money or time or too dangerous.

For example, you may not want to wait a year or more to see telltale symptoms of a progressive illness, such as diabetes. Instead, you might build a classification system for early indicators of the disease that will assign risk in the future without having to wait until irreversible damage has possibly occurred. By knowing that a patient is prediabetic or likely to develop diabetes, doctors can provide options for prevention or early treatment.

> **TIP** Classification is often a way of determining the value of an expensive variable by exploiting a combination of cheap variables.

In related situations, a classification system can reduce physical danger. It's preferable to be able to correlate data obtained by noninvasive means, such as cognitive tests or PET scans, to determine the existence of Alzheimer's disease than to perform direct testing through autopsy on the brain, especially while the patient is still alive.

So classification can be useful. But why use Mahout instead of some other software?

13.3 How classification works

There are two main phases involved in building a classification system: the creation of a model produced by a learning algorithm, and the use of that model to assign new data to categories. The selection of training data, output categories (the targets), the algorithm through which the system will learn, and the variables used as input are key choices in the first phase of building the classification system.

The basic steps in building a classification system are illustrated in figure 13.2. The figure shows two phases of the classification process: the upper path represents the training of the classification model; the lower path provides new examples to the model, which will assign categories (the target variables) as a way to emulate decisions. Input for the training algorithm consists of example data labeled with known target variables. The target variables are, of course, unknown to the model when using new examples during production. In evaluation, the value of the target variable is known, but it won't be given to the model.

For testing, which isn't shown explicitly in figure 13.2, new examples that were withheld from the training data are presented to the model. The results chosen by the model are compared to known answers for the target variables in order to evaluate the model's performance, a process described in depth in chapter 15.

The terminology used by different people to describe classification is highly variable. For consistency, we've limited the terms used for key ideas in the book. Many of these terms are listed in table 13.2. Note, in particular, the relationship between a

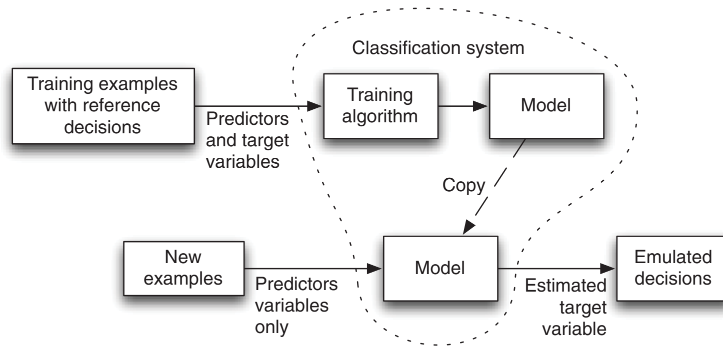


Figure 13.2 How a classification system works. Inside the dotted lasso is the heart of the classification system—a training algorithm that trains a model to emulate human decisions. A copy of the model is then used in evaluation or in production with new input examples to estimate the target variable.

record and a field: the *record* is the repository for values related to a training example or production example, and a *field* is where the value of a feature is stored for each example. The key ideas listed in this table are discussed in the subsections that follow.

Table 13.2 Terminology for the key ideas in classification

Key idea	Description
Model	A computer program that makes decisions; in classification, the output of the training algorithm is a model.
Training data	A subset of training examples labeled with the value of the target variable and used as input to the learning algorithm to produce the model.
Test data	A withheld portion of the training data with the value of the target variable hidden so that it can be used to evaluate the model.
Training	The learning process that uses training data to produce a model. That model can then compute estimates of the target variable given the predictor variables as inputs.
Training example	An entity with features that will be used as input for learning algorithm.
Feature	A known characteristic of a training or a new example; a feature is equivalent to a characteristic.
Variable	In this context, the value of a feature or a function of several features. This usage is somewhat different from the use of *variable* in a computer program.
Record	A container where an example is stored; such a record is composed of fields.
Field	Part of a record that contains the value of a feature (a variable).
Predictor variable	A feature selected for use as input to a classification model. Not all features need be used. Some features may be algorithmic combinations of other features.
Target variable	A feature that the classification model is attempting to estimate: the target variable is categorical, and its determination is the aim of the classification system.

In the following subsections, we look more closely at what a model is and how it's trained, tested, and used in production. You'll learn how to distinguish predictor variables from target variables, understand how to correctly make use of records, fields, and values, and see the four forms of values that variables can take. In addition, you'll see that Mahout classification is an example of supervised learning.

13.3.1 Models

A classification algorithm learns from examples in a process known as *training*. The output of this training process is called a *model*. This model is a function that can then be applied to new examples in order to produce outputs that emulate the decisions that were made in the original examples. These emulated decisions are the end product of the classification system.

> **NOTE** The model produced by the training algorithm is itself effectively a computer program.

You saw in figure 13.2 how training examples, complete with desired output values, are given to the learning part of the classification system to create a model. This model, which itself can be viewed as a computer program that makes decisions, is then copied and given new examples as input. The intention is that the model will classify the new examples, emulating the training examples used as input.

13.3.2 Training versus test versus production

In practice, training examples are typically divided into two parts. One part, known as the *training data*, consists of 80–90 percent of the available data. The training data is used in training to produce the model. A second part, called the *test data*, is then given to the model without telling it the desired answers, although they're known. This is done in order to compare the output of the model with the desired output.

Once the model performs as desired, it's typically put into production and given additional examples to classify, for which the correct decision is unknown. Production results are usually not 100 percent accurate, but the quality of the output should stay consistent with what was achieved during the test runs unless the quality of the input data or external conditions changes.

It's common to take samples of these production examples over time in order to make more training data, so that updated versions of the model can be produced. This sampling of production examples is important if external conditions might change, causing the quality of the decisions made by the model to degrade over time. In that case, a new model can be produced and the quality of the decisions can be improved or restored to original levels.

13.3.3 Predictor variables versus target variable

A *variable* is a value for a feature or characteristic of an example. The value can be derived by measurement or by computation. The ultimate goal of a classifier is to

estimate a categorical answer to a specific question, and this answer is known as the *target variable*. In production, the target variable is the value being sought for each new example. In training, the target variable is known for historical data (training data) and is used to train the model.

In classification, the *predictor variables* are the clues given to the model so it can decide what target variable to assign to each example. Predictor variables used for classification are also known as *input variables* or *predictors*.

The characteristics of the examples to be classified can also be called *features*; the process of describing a particular characteristic in a way that can be used by a classification system is known as *feature extraction*. If a feature is chosen to be used as input for the model, the value of that feature would then be thought of as a predictor variable.

Recall the situation of the shopper buying wine for dinner. Attributes of the wine, such as "color," "good with steak," "good with fish," "good with pizza," and other attributes such as price are used by the shopper to make a decision to buy or not to buy. In the purchase decision, these features of the wine are analogous to predictor variables input to the model in machine classification. In this analogy, the decision has a binary target variable: buy or don't buy. Only one of these choices will be selected for each bottle.

> **NOTE** Both the target and predictor variables are given to the learning algorithm during training. In contrast, during testing and in production, only the predictor variables are available to the model.

In preparing the classification system, the bulk of the historical examples used as training data will be labeled with the value of the target variable. During testing, the target variables for the test data will be cloaked. A comparison of the estimated target values produced by the model with the known values for each test example will reveal the accuracy of the classification model.

For classification, the field representing the target variable must have a categorical value. The value for predictor variables can be continuous, categorical, text-like, or word-like. You can see the differences between these types of values (continuous, categorical, text-like or word-like) in section 13.3.5. Often the target variable is a binary categorical variable, meaning it has only two possible values. Spam detection is an example of a classification system with a binary categorical variable: a message is spam or not spam. When there could be more than two possible values for the target variable, it's called multi-class classification. An example of multi-class classification using data from 20 newsgroups is presented in chapter 14 (sections 14.4 and 14.6).

13.3.4 Records, fields, and values

The examples that comprise the input for classification algorithms are normally represented in the form of records. The A *record* can be thought of as a repository for the values of an example (whether a training example or a new example used in production). Records typically consist of a collection of named fields, each of which has a value.

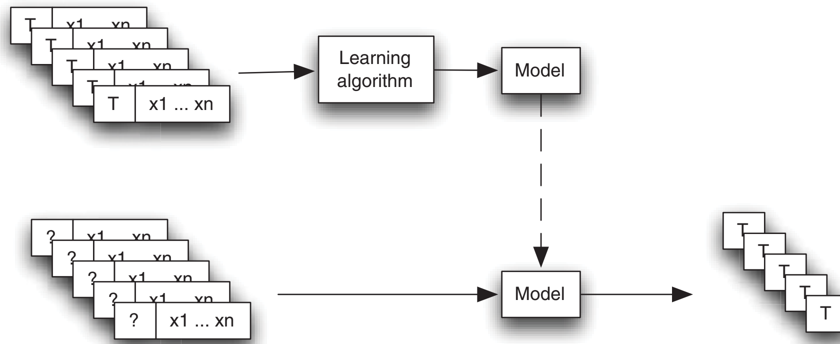


Figure 13.3 Input and output of a classification model: target variables (T) and predictor variables (x1 ... xn). Notice that T is included with the input data during training but it's absent from the input data in production (as indicated by?).

The record for each training example will have fields to store values for the target variable and for one or more features chosen as predictor variables, as illustrated in figure 13.3. The field for the target variable in a production example is, of course, the unknown value that's being sought by the classification algorithm. The output of the model in production is the value estimated for the target variable, denoted in the figure by *T*. There is one output for each input example.

Although target values can only be categorical, several types of values can represent features used as predictor variables. We compare them next.

13.3.5 The four types of values for predictor variables

Predictor variables serve as the input for the learning algorithm, but which variables are useful depends in part on the type of values used to represent them. The distinction between different types of values for variables isn't just an academic exercise. Your ability to correctly identify the types of values you have available will improve the success of your classification system.

There are four common types of values of predictor variables: continuous, categorical, word-like, and text-like, as described in table 13.3.

Table 13.3 Four common types of values used to represent features

Type of value	Description
Continuous	This is a floating-point value. This type of value might be a price, a weight, a time, or anything else that has a numerical magnitude and where this magnitude is the key property of the value.
Categorical	A categorical value can have one of a set of prespecified values. Typically the set of categorical values is relatively small and may be as small as two, although the set can be quite large. Boolean values are generally treated as categorical values. Another example might be a vendor ID.

Table 13.3 Four common types of values used to represent features *(continued)*

Type of value	Description
Word-like	A word-like value is like a categorical value, but it has an open-ended set of possible values.
Text-like	A text-like value is a sequence of word-like values, all of the same kind. Text is the classic example of a text-like value, but a list of email addresses or URLs is also text-like.

Distinguishing between continuous and categorical values is often difficult. It's a common mistake to treat numeric identifier codes as continuous values when they should really be considered categorical values. Consider, for instance, this list of values that represents the first three digits of ZIP codes: 757, 415, 215, 809, and 446. These values look numerical, and you might be tempted to consider them continuous, but they better fit the description of categorical because each one is from a set of prespecified values. Treating a value as continuous when it's really categorical (or vice versa) can seriously impair the accuracy of a classifier.

> **TIP** Things that look like integers aren't necessarily continuous. A key test is to imagine adding two values together or taking the log or square root of a value. If this doesn't make any sense, then you probably have a categorical value rather than a continuous one. Another hint is that anything with an associated unit of measurement is usually a continuous variable.

A word-like value is an extension of categorical values to the point where there are many possible values. The distinction between these two types hinges on how many values a feature could have. Even if a feature could have many values, is there a finite group of prespecified values? In cases where you can't say how many possible values there are, the variable likely is word-like. It's often difficult to decide whether something should be treated as word-like or categorical. For instance, the combined make and model of a car might seem categorical, but if there is a good chance that a new car make or model could appear at any time, treating such a value as word-like might be a better choice.

In the occasional situation where categorical values have thousands or more possible values, they're often better treated as word-like. If you already have many possible values, it's likely that the set of values might not be as fixed as you might think. Occasionally, categorical values might have some ordering, but the classification algorithms in Mahout don't currently pay attention to such ordering.

If you have a variable whose values can be viewed as collections of word-like values, you should consider it to be text-like, even if only a few examples might have more than one such value. Text-like values are often stored as strings and thus need a tokenizer that can convert the string representation as a sequence of word-like values.

TIP A text-like variable can often be spotted by the fact that the components of the value can appear in almost any combination. For this reason, a variable that could have the values *General Motors*, *Ford*, or *Chrysler* isn't text-like, even though *General Motors* appears to have two words in it. That's because values like *General Chrysler Ford* don't make sense.

Keep in mind the descriptions in table 13.3 as you examine table 13.4, which shows examples from different data sources. The first value, an email address, is a single entity from an open-ended set and thus is word-like. The values for spam words or non-spam words comprise a list of words (or possible words) and are considered text-like.

Table 13.4 Sample data that illustrates all four value types. These examples are typical of features of email data.

Name	Type	Value
from-address	Word-like	`George <george@fumble-tech.com>`
in-address-book?	Categorical (`TRUE`, `FALSE`)	`TRUE`
non-spam-words	Text-like	`Ted`, `Mahout`, `User`, `lunch`
spam-words	Text-like	`available`
unknown-words	Continuous	`0`
message-length	Continuous	`31`

NOTE Details of which types of predictor values are used by a particular algorithm are covered in chapter 14, section 14.5.

One of the distinctions between classification and other Mahout capabilities, such as clustering or recommendation, is based on whether learning is supervised or unsupervised. The next section explains this terminology.

13.3.6 Supervised versus unsupervised learning

Classification algorithms are related to, but still quite different from, clustering algorithms such as the k-means algorithm described in previous chapters. Classification algorithms are a form of supervised learning, as opposed to unsupervised learning, which happens with clustering algorithms. A supervised learning algorithm is one that's given examples that contain the desired value of a target variable. Unsupervised algorithms aren't given the desired answer, but instead must find something plausible on their own.

Supervised and unsupervised learning algorithms can often be usefully combined. A clustering algorithm can be used to create features that can then be used by a learning algorithm, or the output of several classifiers can be used as features by a clustering algorithm. Moreover, clustering systems often build a model that can be used to categorize new data. This clustering system model works much like the model produced by a classification system. The difference lies in what data was used to produce the model. For classification, the training data includes the target variables; for clustering, the training data doesn't include target variables.

You now have a fundamental idea of what classification is, and you know about the advantages of using Mahout for classifying very large data sets. It's time to put these ideas into practice.

13.4 Work flow in a typical classification project

Regardless of the differences between particular projects, there's a pretty standard set of steps that need to be followed to build and run a classification system. In this section, we provide an overview of the typical workflow in classification projects: training the model, evaluating and adjusting the model to an acceptable level, and running the system in production. We look at an example using synthetic data to illustrate some of the steps in the workflow. We then look at a more complete example of this type in section 13.5 to complete a classification project using Mahout.

As a prerequisite to any project, you need to identify the information you're seeking (the target variable) as a feature that can be stored in appropriate form in a record and that meets the overall goal, such as identifying a fraudulent financial transaction. Sometimes you'll have to adjust what you choose as a target variable for pragmatic reasons, such as the cost of training, privacy issues, and so on. You also need to know what data is available for training and what new data will be available for running the system in order to design a useful classifier.

> **NOTE** Commonly, most of the effort required to build a classifier is spent inventing and extracting useful features. It's also common to massively underestimate just how much work this step will involve.

The general steps in developing a classification project are listed in table 13.5. In each stage of the process, the accuracy of the system and the usefulness of the output largely reflect the original selection of features to be used as predictors and the categories chosen for the target variable. There isn't a single, correct selection for each of these choices: a variety of approaches and adjustments are possible and experimenting with several is usually required to build an efficient and robust system.

Table 13.5 Workflow in a typical classification project

Stage	Step
1. Training the model	Define target variable. Collect historical data. Define predictor variables. Select a learning algorithm. Use the learning algorithm to train the model.
2. Evaluating the model	Run test data. Adjust the input (use different predictor variables, different algorithms, or both).
3. Using the model in production	Input new examples to estimate unknown target values. Retrain the model as needed.

Each step in the process of training and evaluating the model requires a balance between selecting features for variables that are practical and affordable in cost and time, and getting an acceptable level of accuracy in the values estimated by the model.

In the following sections, we discuss what happens during the three stages of classification.

13.4.1 Workflow for stage 1: training the classification model


No. 12

In the first stage of a classification project, you need to have a target variable in mind, and this will help you select both appropriate historical data for the training process and a useful learning algorithm. These decisions go hand in hand.

In the following discussion, we examine a few specific ways in which your choice of features will influence which Mahout learning algorithms will work in your classifier.

DEFINE CATEGORIES FOR THE TARGET VARIABLE

Defining your target variable involves defining the categories for the target variable. The target variable can't have an open-ended set of possible values. Your choice of categories, in turn, affects your choices for possible learning algorithms, because some algorithms are limited to binary target variables. Having fewer categories for the target variable is definitely preferable, and if you can pare the categories down to just two values, you'll have a few more learning algorithms available to you.

If the target variable resists your efforts to reduce it to a simple form, you may need to build several classification systems, each deriving some aspect of your desired target variable. In your final system, you could then combine the outputs of these classification systems to produce the fully nuanced decision or prediction that you need.

COLLECT HISTORICAL DATA

The source of historical data you choose will be directed in part by the need to collect historical data with known values for the target variable. For some problems, there will be difficulties in determining exactly what the value of the target variable should be.

The old saying about garbage in leading to garbage out is particularly apt here, and great care should be taken to make sure that the value of the target variable is accurate for the historical data. Many models have been built that faithfully replicate a flawed target variable or data collection process, and you don't want to build another one.

DEFINE PREDICTOR VARIABLES

After you have selected a useful target variable and have defined the set of values it can take on, you need to define your predictor variables. These variables are the concrete encoding of the features extracted from the training and test examples.

Once again, you'll need to examine the source of your examples to make certain that they have useful features with values that can be stored in the appropriate format for their fields in the records. As you saw in figure 13.3, the predictor variables appear in records for the training and test data and for the production data.

> **WARNING** A common mistake in selecting and defining predictor variables is to include what is known as a *target leak* in the predictor variables.

An important consideration in defining the predictor variables is to avoid setting up a target leak. A *target leak*, as the name implies, is a bug that involves unintentionally providing data about the target variable in the selection of the predictor variables. This bug isn't to be confused with intentionally including the target variable in the record of a training example.

Target leaks can seriously affect the accuracy of the classification system. This problem can be relatively obvious when you tell the classifier that the target variable is a predictor, an error that seems obvious but that happens distressingly often.

Target leaks can be subtler as well. Suppose in building a spam detector, you keep the spam and non-spam examples in separate files and then label the examples sequentially within each file. Examples in the same file would have consecutive sequence numbers within a limited range and would have identical spam and non-spam statuses. With this type of target leak, many classification algorithms will quickly figure out the sequence number ranges that correspond to spam and to non-spam, and will settle on that one feature because it appears (in the training data) to be completely reliable. There lies the problem. When you give new data to a model that's based almost entirely on sequence numbers, the best that the model could do is throw up its hands, because the sequence numbers in the new examples would be outside the previously seen ranges. More likely, the model would attribute new examples to whichever range had the highest sequence numbers.

Target leaks can be quite insidious and difficult to find. The best advice is to be suspicious of models that produce results that seem too good to be true.

EXAMPLE 1: USING POSITION AS A PREDICTOR VARIABLE

Let's look at a simple example using synthetic data that illustrates how you can select predictor variables that will let a Mahout model accurately predict the desired target variable. The data displayed in figure 13.4 is a collection of historical data. Suppose you're in search of filled shapes—color-fill is your target variable. What features will best serve as predictor variables for this problem?

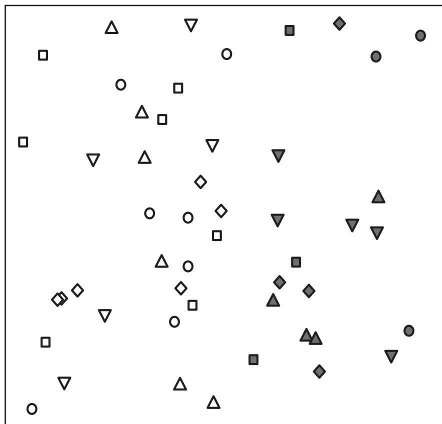


Figure 13.4 Using position to classify color-fill: the target variable in this historical data is color-fill, and the features to be considered as predictor variables include shape and position. Position looks more promising as a predictor—the horizontal (*x*) coordinate may be sufficient. Shape doesn't seem to matter.

For a classification problem, you must select categories for your target variable, which in this case is color-fill. Color-fill is clearly categorical with two possible values that you could label *filled* and *unfilled*. Or you could set up the target variable as a question with *yes* and *no* responses: "Is the item filled?"

Don't take anything for granted: inspect the historical data to make certain it includes the target variable. Here it's obvious that it does (good!), but in real world situations, it isn't always obvious.

Now you must select the features for use as predictor variables. What features can you describe appropriately? Color-fill is out (it's the target variable) but you can use position or shape as variables. You could describe position by the x and y coordinates. From a table of the data, you can create a record for each example that includes fields for the target variable and for the predictor variables you're considering. A record for two of the training examples is shown in figure 13.5. (The complete data for this sample can be found in the examples module of the Mahout source code.)

If you once again examine the historical data depicted in figure 13.4, you'll likely see that although the training data has both position and shape information, the *x* coordinate suffices to make the differentiation between filled and unfilled symbols. Shape has no utility in trying to determine whether a shape should be labeled as filled or not, nor does the *y* coordinate.

> **TIP** Not all features are useful for every classification problem. The specific situation and question you're asking will determine which features make effective predictor variables. Learning about your data is crucial for success.

As you design your classification system, you'll select those features that your experience suggests are most likely to be effective, and the accuracy of your model will tell you if you have chosen wisely. It isn't necessary to exclude all features that aren't useful in differentiation, but the fewer redundant or useless features you include, the more likely it is that your classifier will produce accurate results. For example, in the data shown in figure 13.4, you would do best to use only the *x* coordinate field for your final model and ignore the *y* coordinate and shape.

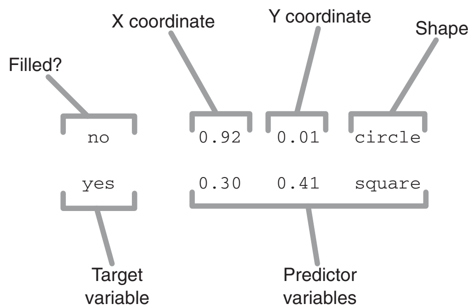


Figure 13.5 Two records for training data. The records for data shown in figure 13.4 have fields that store the values of the target variable and fields for the values of the predictor variables. In this case, values relating to position (*x, y* coordinates) and shape have been included for each of the two examples.

EXAMPLE 2: DIFFERENT PREDICTOR VARIABLES ARE NEEDED WITH DIFFERENT DATA

Just because a new collection of data exhibits the same features as a previous collection doesn't mean you should use the same features as before for your predictor variables. This idea is illustrated in figure 13.6. Here you can see another collection of historical data that has the same features as the previous group. But in this situation, neither the *x* nor *y* coordinates seem helpful in predicting whether a symbol is filled or unfilled. Position is no longer useful, but shape is now a useful feature.

In this example, the feature chosen for a predictor variable (shape) has three values (round, triangle, square). Had it been necessary, you could also have used orientation to further differentiate shape (square versus diamond; upward-pointing triangle versus downward-pointing triangle).

SELECT A LEARNING ALGORITHM TO TRAIN THE MODEL

In any project, you must select which algorithm to use, taking into consideration parameters such as the amount of training data, the characteristics of the predictor variables, and the number of categories in the target variable. Mahout classification algorithms include naive Bayes, complementary naive Bayes, stochastic gradient descent (SGD), and random forests. The use and behavior of naive Bayes, complementary naive Bayes, and SGD are described in detail in chapter 14 (section 14.5).

Even in these simplistic examples, different algorithms have different advantages. In example 1, the training algorithm should make use of the *x* coordinate position to determine color-fill. In example 2, shape is more useful. The *x* position of a point is, however, a continuous variable, and some algorithms can make good use of continuous variables. In Mahout, SGD and random forests do this. Other algorithms can't use continuous variables, such as naive Bayes and complementary naive Bayes.

The improved speed of a parallel algorithm is obvious when you have an extremely large data set. Why, then, would you ever use the nonparallel (sequential) algorithm for classification? The answer is simply that there is a trade-off. The parallel algorithm does have considerable overhead, meaning that it takes a while to "find the car keys

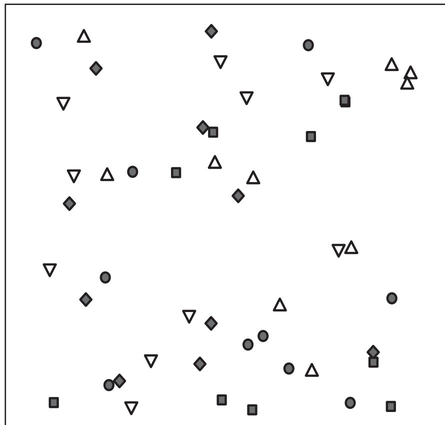


Figure 13.6 Using shape to classify color-fill: this data displays the same features (shape and position) as figure 13.4, but different predictor variables are needed to predict the target variable (color-fill). Here, position isn't readily helpful, but shape is useful.

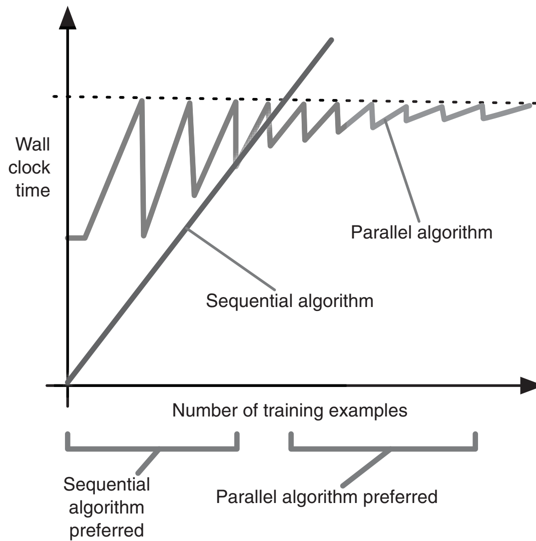


Figure 13.7 Comparing two Mahout classifier algorithms. In both cases, the algorithms are scalable, as depicted in this conceptual graph.

and get down the driveway" before it even begins processing any examples. Where you can afford the time, as with a medium-sized data set, the sequential algorithm may not only be sufficient but preferred. This trade-off is illustrated in figure 13.7, where the runtimes for hypothetical sequential and parallel scalable algorithms are compared.

Over the lower range of project sizes, the sequential algorithm provides a time advantage. In the midrange of size, both perform equally well. As the number of training examples becomes very large, the parallel algorithm outperforms the sequential one.

As explained for figure 13.1, the jagged shape of the parallel algorithm shows the times at which declining performance is revived by the addition of new machines.

USE LEARNING ALGORITHM TO TRAIN THE MODEL

After an appropriate set of target variables and predictor variables have been identified and encoded, and a learning algorithm has been chosen, the next step in training is to run the training algorithm to produce a model (these steps were depicted in figure 13.3). This model will capture the essence of what the learning algorithm is able to discern about the relationship between the predictor variables and the target variable.

For example 1, the model would be something like the following pseudo-code:

```
if (x > 0.5) return FILLED;
else return UNFILLED;
```

Of course, a real model produced by a learning algorithm isn't implemented this way, but this illustrates how simple a rule can be and still work in this example.

13.4.2 Workflow for stage 2: evaluating the classification model

An essential step before using the classification system in production is to find out how well it's likely to work. To do this, you must evaluate the accuracy of the model and make large or small adjustments as needed before you begin classification.

Evaluating a trained model is rarely a simple process. Chapter 16 provides a detailed explanation of how to go about evaluating and fine-tuning a model in preparation for running the system in production. In the step-by-step example in section 13.5, you'll see the beginnings of how evaluation works.

13.4.3 Workflow for stage 3: using the model in production

Once the model's output has reached an acceptable level of accuracy, classification of new data can begin. The performance of the classification system in production will depend on several factors, one of the most important being the quality of the input data. If the new data to be analyzed has inaccuracies in the values of predictor variables, or if the new data isn't an appropriate match to the training data, or if external conditions change over time, the quality of the classification model's output will degrade. In order to guard against this problem, periodic retesting of the model is useful, and retraining may be necessary.

To do retesting, you need to gather new examples and verify or generate the value of the target variable. Then you can run these new examples as test data, just as in the second stage, and compare these results with the results of the original withheld data used in testing. If these new results are substantially worse, then something has changed and you need to consider retraining the model.

One common reason retraining becomes necessary is that the incorporation of the model into your system may change the system substantially. Imagine, for example, what happens if a large bank deploys a very effective fraud-detection system. Within a few months or years, the fraudsters will adapt and start using different techniques that the original model can't detect as well. Even before these new fraud methods become a significant loss, it's very important that the bank's modelers detect the loss in accuracy and respond with updated models.

If the performance of a classification system has diminished, it's probably not necessary to discard the approach. Retraining using new and more apropos training data may be sufficient to update the model. In addition, some adjustments to the training algorithm can be useful. Ongoing evaluation is a part of retraining, as described in more detail in the discussion of model updates in chapter 16 (section 16.4.1).

13.5 Step-by-step simple classification example

You've now learned the basics of what classification is, and how the workflow progresses in a typical project. You've seen the importance of carefully selecting which features to use as predictor variables based on the question you're asking and which target variables you're estimating.

Although Mahout is intended for extremely large data sets, it's helpful to start with a simple example as you familiarize yourself with Mahout. In this section, you'll have a chance to put Mahout into action as a classifier for a small practice data set. We describe the data involved, how to train a model, and how to tune the classifier a bit for better accuracy.

13.5.1 The data and the challenge

In this section, you'll train a classification model to assign color-fill to a collection of synthetic data similar to those in the previous sections. In the training data, the target variable once again will be color-fill, having two categories: *filled* and *not filled*. In this data, position is the key to predicting color-fill, and you'll use a Mahout classification algorithm called stochastic gradient descent (SGD) to train the model.

The historical data for this do-it-yourself practice project is depicted in figure 13.8 and is included as part of the Mahout distribution. To help you get started with classification, Mahout has several such data sets built into the JAR file that contains the example programs. We refer to this data set as the *donut data*.

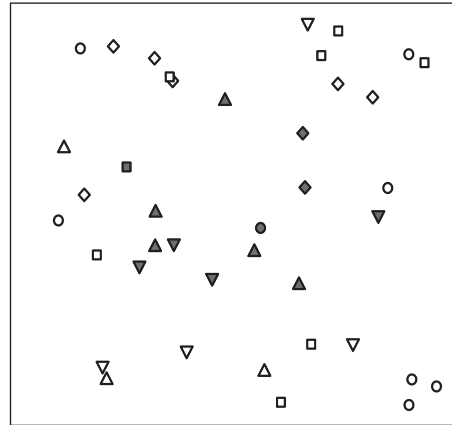


Figure 13.8 Example 3 data for a simple classification project: the donut data. What features are useful for finding filled points?

For this simple classification project, the target variable is color-fill, and your goal is to build a system capable of finding filled points.

13.5.2 Training a model to find color-fill: preliminary thinking

As you consider how to train a model for your project, think about what will work as predictor variables. In the example discussed in section 13.3.1 and depicted in figure 13.4, position rather than shape was key. A cursory inspection of figure 13.8 tells you that for this simple collection of data, position is once again key. But you may suspect that neither x nor y coordinates, nor a combination of the two, are likely to be sufficient to predict the location of filled points for this data set.

For a feature such as position, there is more than one way to consider it as a variable. Perhaps position defined as distance to a particular fixed point such as A, B, or C would be useful, as shown in figure 13.9.

Although x and y coordinates are sufficient to specify any particular point, they aren't useful features to predict whether a point is filled. Adding the distances to points A, B, and C restates position in a way that makes it easier to build an accurate classifier for this data. Distance to C is promising, but you may choose to include

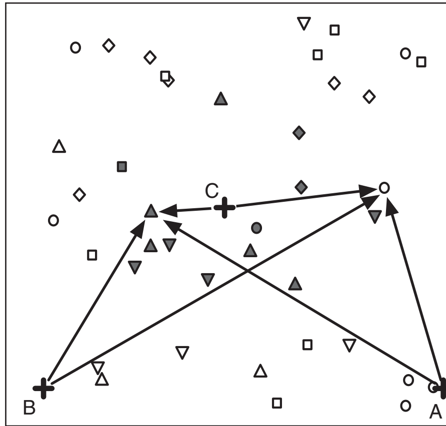


Figure 13.9 If position is the feature used as the predictor variable for this donut data set, you must decide how best to designate position to achieve your particular goal—to identify filled points. The distance to C looks the most useful.

additional predictor variables in your system. The details for doing this are described in the following sections.

13.5.3 *Choosing a learning algorithm to train the model*

Mahout provides a number of classification algorithms, but many are designed to handle very large data sets and as a result can be a bit cumbersome to use, at least to start. Some, on the other hand, are easy to get started with because, though still scalable, they have low overhead on small data sets.

One such low overhead method is the stochastic gradient descent (SGD) algorithm for logistic regression. This algorithm is a sequential (nonparallel) algorithm, but it's fast, as was shown in the conceptual graph in figure 13.9. Most importantly for working with large data, the SGD algorithm uses a constant amount of memory regardless of the size of the input.

START RUNNING MAHOUT

First, download and unpack Mahout from mahout.apache.org. Assuming that you put the unpacked directory name in the `MAHOUT_HOME` environment variable, you can list the commands available with the command-line tool in Mahout:

```
$ $MAHOUT_HOME/bin/mahout
An example program must be given as the first argument.
Valid program names are:
  canopy: : Canopy clustering
  cat : Print a file or resource as the logistic regression models would see
     it
...
  runlogistic : Run a logistic regression model against CSV data
  ...
 trainlogistic : Train a logistic regression using stochastic gradient
     descent
 ...
```

The commands that we'll be most interested in here are the `cat`, `trainlogistic`, and `runlogistic` commands.

CHECK MAHOUT'S BUILT-IN DATA

To help you get started with classification, Mahout includes several data sets as resources in the JAR file that contains the example programs (see the examples/src/main/resources/ directory). When you specify an input to the SGD algorithm, if the input you give doesn't specify an existing file, the training or test algorithm will read the resource of that name instead, if a resource of that name exists.

To list the contents of any of these resources, use the cat command. For instance, this command will dump the contents of the donut data set illustrated in figure 13.8:

```
$ bin/mahout cat donut.csv
"x","y","shape","color","k","k0","xx","xy","yy","a","b","c","bias"
0.923307513352484,0.0135197141207755,21,2,4,8,0.852496764213146,...,1
0.711011884035543,0.909141522599384,22,2,3,9,0.505537899239772,...,1
...
0.67132937326096,0.571220482233912,23,1,5,2,0.450683127402953,...,1
0.548616112209857,0.405350996181369,24,1,5,3,0.300979638576258,...,1
0.677980388281867,0.993355110753328,25,2,3,9,0.459657406894831,...,1
$
```

Note that most of the contents aren't shown, as indicated by the ellipsis (...). This file consists of a first line that specifies the names of the fields in the data, followed by more lines that contain the data itself. As you can see, the original predictor variables that we talked about are included: *x*, *y*, shape, and color. The values of *x* and *y* are numerical values in the range [0, 1] whereas shape is an integer in the range [21 ... 25]. The color is an integer that's always 1 or 2. Because subtracting a triangle from a square makes no sense, we should consider the shape variable to be categorical. Likewise, the color values should be considered numerical codes for a categorical variable with two values.

In addition to x, y, shape, and color, there are additional variables present in this data to allow you to experiment with different classification schemes. These variables are described in table 13.6.

Table 13.6 Fields used in the donut.csv data file

Variable	Description	Possible values
x	The *x* coordinate of a point	Numerical from 0 to 1
y	The *y* coordinate of a point	Numerical from 0 to 1
shape	The shape of a point	Shape code from 21 to 25
color	Whether the point is filled or not	1=empty, 2=filled
k	The k-means cluster ID derived using only *x* and *y*	Integer cluster code from 1 to 10
k0	The k-means cluster ID derived using *x*, *y*, and color	Integer cluster code from 1 to 10
xx	The square of the *x* coordinate	Numerical from 0 to 1
xy	The product of the *x* and *y* coordinates	Numerical from 0 to 1
yy	The square of the *y* coordinate	Numerical from 0 to 1

Table 13.6 Fields used in the donut.csv data file *(continued)*

Variable	Description	Possible values
`a`	The distance from the point (0,0)	Numerical from 0 to $\sqrt{2}$
`b`	The distance from the point (1,0)	Numerical from 0 to $\sqrt{2}$
`c`	The distance from the point (0.5,0.5)	Numerical from 0 to $(\sqrt{2})/2$
`bias`	A constant	1

BUILD A MODEL USING MAHOUT

You can build a model to determine the color field from the *x* and *y* features with the following command:

```
$ bin/mahout trainlogistic --input donut.csv \
       --output ./model \
       --target color --categories 2 \
       --predictors x y  --types numeric \
       --features 20 --passes 100 --rate 50
...
color ~ -0.157*Intercept Term + -0.678*x + -0.416*y
      Intercept Term -0.15655
                   x -0.67841
                   y -0.41587
...
```

This command specifies that the input comes from the resource named donut.csv, that the resulting model is stored in the file ./model, that the target variable is in the field named `color` and that it has two possible values. The command also specifies that the algorithm should use variables `x` and `y` as predictors, both with numerical types. The remaining options specify internal parameters for the learning algorithm.

> **NOTE** The actual output you see will probably only agree to one significant figure because the logistic regression training algorithm used is somewhat randomized.

A list of all of the command-line options for the `trainlogistic` program is shown in table 13.7.

Table 13.7 Command-line options for the `trainlogistic` program

Option	What it does
`--quiet`	Produces less status and progress output.
`--input <file-or-resource>`	Uses the specified file or resource as input.
`--output <file-for-model>`	Puts the model into the specified file.
`--target <variable>`	Uses the specified variable as the target.
`--categories <n>`	Specifies how many categories the target variable has.
`--predictors <v1> ... <vn>`	Specifies the names of the predictor variables.

Table 13.7 Command-line options for the `trainlogistic` program *(continued)*

Option	What it does
`--types <t1> ... <tm>`	Gives a list of the types of the predictor variables. Each type should be one of `numeric`, `word`, or `text`. Types can be abbreviated to their first letter. If too few types are given, the last one is used again as necessary. Use `word` for categorical variables.
`--passes`	Specifies the number of times the input data should be reexamined during training. Small input files may need to be examined dozens of times. Very large input files probably don't even need to be completely examined.
`--lambda`	Controls how much the algorithm tries to eliminate variables from the final model. A value of 0 indicates no effort is made. Typical values are on the order of 0.00001 or less.
`--rate`	Sets the initial learning rate. This can be large if you have lots of data or use lots of passes because it's decreased progressively as data is examined.
`--noBias`	Eliminates the intercept term (a built-in constant predictor variable) from the model. Occasionally this is a good idea, but generally it isn't since the SGD learning algorithm can usually eliminate the intercept term if warranted.
`--features`	Sets the size of the internal feature vector to use in building the model. A larger value here can be helpful, especially with text-like input data.

13.5.4 Improving performance of the color-fill classifier

Now that you have trained your first classification model, you need to determine how well it's performing the task of estimating color-fill condition. This evaluation stage isn't just a way to assign a grade for your project. It's your means of assessing and improving your classifier for excellent performance.

EVALUATE THE MODEL

Remember that this problem has the filled points completely surrounded by unfilled points, which makes it impossible for a simple model, such as is produced by the SGD algorithm using only x and y coordinates, to classify points accurately. Indeed, the linear equation underlying this model can only produce a negative value, which, in the context of logistic regression, is guaranteed to never produce a score greater than 0.5. As we shall see, this first model isn't very useful.

After the model is trained, you can run the model on the training data again to evaluate how it does (even though we know it won't do well).

```
$  bin/mahout runlogistic --input donut.csv --model ./model \
        --auc --confusion
AUC = 0.57
confusion: [[27.0, 13.0], [0.0, 0.0]]
...
```

The output here contains two values of particular interest. First, the AUC value (an acronym for *area under the curve*—a widely used measure of model quality) has a value of 0.57. AUC can range from 0 for a perfectly perverse model that's always exactly wrong to 0.5 for a model that's no better than random to 1.0 for a model that's perfect. The value here of 0.57 indicates a model that's hardly better than random.

To understand why this model performs so poorly, you can look to a *confusion matrix* for a clue. A confusion matrix is a table that compares actual results with desired results. All examples at the default score threshold of 0.5 are classified as being unfilled. This allows the classifier to be correct two-thirds of the time (27 out of 40 times), but it only manages to be correct marking everything as unfilled. This model gets the right answer much of the time, but only in the way that a stopped clock is right twice a day.

> **NOTE** Along with other measurement tools, the confusion matrix is described in more detail in chapter 15.

The options accepted by the `runlogistic` command are summarized in table 13.8.

Table 13.8 Command-line options for the `runlogistic` program

Option	What it does
`--quiet`	Produces less status and progress output.
`--auc`	Prints AUC score for model versus input data after reading data.
`--scores`	Prints target variable value and scores for each input example.
`--confusion`	Prints confusion matrix for a particular threshold (see `--threshold`).
`--input <input>`	Reads data records from specified file or resource.
`--model <model>`	Reads model from specified file.

BUILD A MORE INTERESTING MODEL

You can get more interesting results if you use additional variables for training. For instance, this command allows the model to use `x` and `y` as well as `a`, `b`, and `c` to build the model:

```
$ bin/mahout trainlogistic --input donut.csv --output model \
         --target color --categories 2 \
         --predictors x y a b c  --types numeric \
         --features 20 --passes 100 --rate 50
...
color ~ 7.07*Intercept Term + 0.58*x + 2.32*y + 0.58*a + -1.37*b + -25.06*c
      Intercept Term 7.06759
                   a 0.58123
                   b -1.36893
                   c -25.05945
                   x 0.58123
                   y 2.31879
...
```

Notice how this model applies a large weight to the `c` variable, as well as a large intercept term. If you ignore the other variables (which isn't entirely appropriate to do, but the intercept and `c` are definitely dominant here), the output of the linear part of this model will reach a maximum value of 11.5 where `c` = 0, and it will drop rapidly into negative territory once `c` > 0.3. This makes a lot of sense geometrically, based on what we know about the problem. This consideration alone is a strong indication that this isn't a stopped-clock model like the one that depended on `x` and `y` alone.

Logistic regression

Logistic regression describes a kind of classification model in which the predictor variables are combined with linear weights and then passed through a soft-limit function that limits the output to the range from 0 to 1. Logistic regression is closely related to other models such as a perceptron (where the soft limit is replaced by a hard limit), neural networks (where multiple layers of linear combination and soft limiting are used) and naive Bayes (where the linear weights are determined strictly by feature frequencies assuming independence). Logistic regression can't separate all possible classes, but in very high dimensional problems or where you can introduce new variables by combining other predictors, this is much less of a problem. The mathematical simplicity of logistic regression allows very efficient and effective learning algorithms to be derived.

TEST AGAIN

If you run this revised model on the training data, you'll get a very different result than with the previous model:

```
$  bin/mahout runlogistic --input donut.csv --model model \
         --auc --confusion
AUC = 1.00
confusion: [[27.0, 0.0], [0.0, 13.0]]
entropy: [[-0.1, -1.5], [-4.0, -0.2]]
```

Now you have a perfect value of 1 for AUC, and you can see from the confusion matrix that the model was able to classify all of the training examples perfectly.

TEST USING NEW DATA

You can run this same model on additional data in the donut-test.csv resource. Because this new data was not used to train the model, there are a few surprises in store for the model.

```
$  bin/mahout runlogistic --input donut-test.csv --model model \
          --auc --confusion
AUC = 0.97
confusion: [[24.0, 2.0], [3.0, 11.0]]
entropy: [[-0.2, -2.8], [-4.1, -0.1]]
```

On this withheld data, you can see that the AUC has dropped to 0.97, which is still quite good. The confusion matrix shows that of the 40 new samples, there are 5

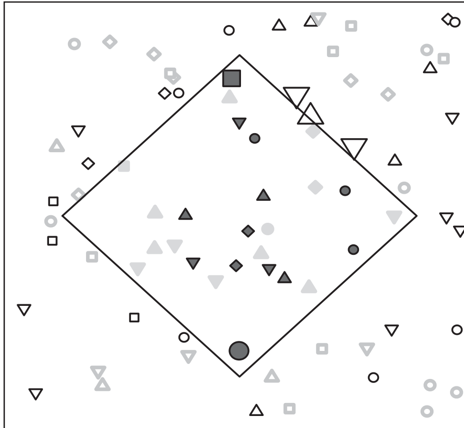


Figure 13.10 Trying the classifier on new data. The original data is shown in a lighter shade, and the new data is shown filled or not. The enlarged symbols are the ones that the model got wrong. The diamond outline shows the region that we actually used to generate the donut data.

incorrectly classified examples, 2 unfilled points that were marked as filled by the model (false positives), and 3 filled points that were marked by the model as unfilled (false negatives). Clearly the use of the additional variables, notably the inclusion of *c*, has helped the model substantially, but there are still a few issues. Figure 13.10 shows what happened.

Notice the large, filled circle near the bottom of the diamond in figure 13.10. This circle is close to training dots that weren't filled, but it's not near any filled training dots. Because the learning algorithm wasn't allowed to know that the correct region had this odd shape, it was nearly impossible for the model to get the right answer.

TRY OTHER MODELS

You can do additional experiments by selecting other sets of variables. For instance, here is what happens when you include `x`, `y`, `a`, and `b`, but exclude `c` from the set of predictor variables:

```
$ bin/mahout trainlogistic --input donut.csv --output model \
           --target color --categories 2 \
           --predictors x y a b  --types n --features 20 \
           --passes 100 --rate 50
...
color ~ 4.634*Intercept Term + -2.834*x + 5.558*y + -2.834*a + -6.000*b
      Intercept Term 4.63396
                   a -2.83439
                   b -5.99971
                   x -2.83439
                   y 5.55773
  ...
$  bin/mahout runlogistic --input donut-test.csv --model model \
             --auc --confusion
AUC = 0.91
confusion: [[27.0, 13.0], [0.0, 0.0]]
   entropy: [[-0.3, -0.4], [-1.5, -0.7]]
$
```

This time, the model does only slightly worse on the withheld data, even though it didn't have access to its favorite variable. This indicates that the information carried by `c` is also available in `x`, `y`, `a`, and `b`.

You can try lots of experiments on the parameters of the learning model, as well. These experiments are the beginning of the evaluation process described in chapter 15.

13.6 Summary

By developing the example in the previous section, you've put to use what you learned throughout this chapter about the first two stages of classification: how to train a model and how to evaluate it and adjust performance. If this were a real system, your model would now be ready for the third stage in classification: deploying it into production in a real-world setting. The basic terminology of classification will now be familiar to you, and you should have a solid understanding of what classification is and how it works.

Remember that an important key to building a successful classifier is to carefully pose your question in specific and simple terms so that it can be answered by a limited list of predetermined categories, the target variables. Be aware that not all features of the input data are equally helpful; you must select the features to be used as predictor variables carefully, and you may need to try many combinations to find which ones are useful for your classifier to perform well. And remember, pragmatism is important.

With these ideas in mind, think back to the example of wine tasting. The goal of your machine-based classifier should be to help you get to dinner on time, not to help you make subtle aesthetic judgments about the finer things in life.

As you continue through chapters 14–17, you'll see how Mahout provides a powerful tool to meet the requirements of extremely large classification systems, particularly those with data sets above 1 million examples. Chapter 14 focuses on how to extract features for input to the learning algorithm as a first step in finding out how to evaluate and tune a trained classifier before deploying it to production.

14 Training a classifier

This chapter covers

- Extracting features from text
- Converting features for Mahout's use
- Training two Mahout classifiers
- Selecting from among Mahout's learning algorithms

This chapter explores the first stage in classification: training the model. Developing a classifier is a dynamic process that requires you to think creatively about the best way to describe the features of your data and to consider how they will be used by the learning algorithm you choose to train your models. Some kinds of data lend themselves readily to classification; others offer a greater challenge, which can be rewarding, frustrating, and interesting all at once.

In this chapter, you'll learn how to choose and extract features effectively to build a Mahout classifier. Feature extraction involves much more than the simplified steps you saw in the examples in chapter 13. Here we go into the details of feature extraction, including how to preprocess raw data into classifiable data and how to convert classifiable data into vectors that can be used by the Mahout classification algorithms. We use a computational marketing problem as an example to show how training data might be extracted from a database.

Once you understand how to get data ready for classification, in section 14.4 you'll build a classifier using a standard data set—20 newsgroups—with the Mahout algorithm known as stochastic gradient descent (SGD).

In section 14.5, we review the characteristics of the various learning algorithms available for use with Mahout classification and help you decide how to choose the ones best suited to particular projects. These two steps, extracting features and choosing the algorithm, go hand in hand in designing and training a classifier, and to help you develop an intuitive sense of the different options, we look at a second step-by-step example using a different learning algorithm, naive Bayes, with the same 20 newsgroups data set.

This chapter starts by explaining how to work with the data in your training examples.

14.1 Extracting features to build a Mahout classifier

Getting data into a form usable by a classifier is a complex and often time-consuming step, and in this section we present an overview of what is involved. You saw a simplified view of the training and use of a classification model in figure 13.2 (in chapter 13); it showed a single step leading from training examples to the training algorithm that trains the classification model. But in reality, the situation is more complex. The important details not mentioned in chapter 13 are shown here, in figure 14.1.

Figure 13.2 showed a single step leading from the training data to the training algorithm. In reality, raw data must be collected and preprocessed to turn it into classifiable training data. The training examples in figure 14.1 are classifiable data—we'll cover the processing of raw data into classifiable data in this chapter and in more detail in chapters 15 and 16.

Once the raw data has been preprocessed into classifiable form, several steps are required to select predictor and target variables and encode them as vectors, the style of input required by Mahout classifiers. Recall from chapter 13 that the values for features to be used as predictor variables can be of four types:

- Continuous
- Categorical
- Word-like
- Text-like

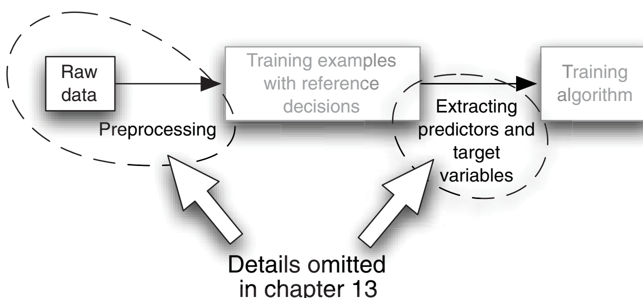


Figure 14.1 An expansion of part of figure 13.2 from the previous chapter. This is what must be done to get raw data ready so that it can be input into a training algorithm.

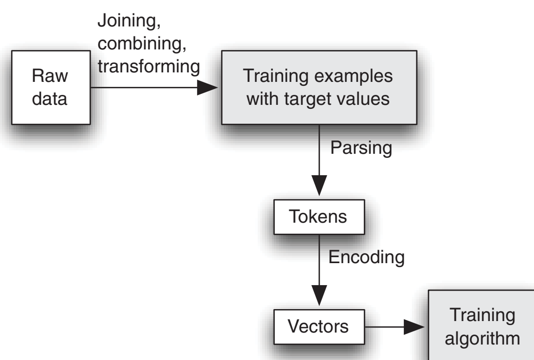


Figure 14.2 Vectors are the format that the classification algorithms require as input. In order to encode data as vectors, raw data must be collected in a single record in classifiable format, ready for parsing and vectorization.

This description of predictor variables is correct on the whole, but it omits the important fact that when the values are presented to any of the algorithms used to train a classifier, they must be in the form of vectors of numbers, either in memory or in a file format that the training algorithm can read.

Compare the simplified sequence depicted in figure 14.1 with the more detailed diagram in figure 14.2. The original data undergoes changes in order to turn it into the vectors required as input for the training algorithm. These changes happen in two phases: preprocessing, to produce classifiable data for use as training examples, and converting classifiable data into vectors.

As figure 14.2 shows, preparing data for the training algorithm consists of two main steps:

1. *Preprocessing raw data*—Raw data is rearranged into records with identical fields. These fields can be of four types: continuous, categorical, word-like, or text-like in order to be classifiable.
2. *Converting data to vectors*—Classifiable data is parsed and vectorized using custom code or tools such as Lucene analyzers and Mahout vector encoders. Some Mahout classifiers also include vectorization code.

The second of these steps has two phases: tokenization and vectorization. For continuous variables, the parsing may be trivial, but for other types of variables, such as categorical, word-like, or text-like variables, vectorization can be involved.

We'll discuss these two steps in detail in subsequent sections of this chapter.

14.2 Preprocessing raw data into classifiable data

The first phase of feature extraction involves rethinking the data and identifying features you could use as predictor variables. You first select a target variable that's in accordance with the goal of classification and pick or drop features to get a mix that's worthy of an initial try. There is no exact recipe: this step calls for intelligent guessing guided by experience, and as you work through the examples in this chapter, you'll be on your way toward getting that experience.

This section offers a brief overview of how to preprocess data. The steps include collecting or rearranging data into a single record, and gleaning secondary meanings from raw data (such as converting a ZIP code to a three-digit code or using a birth date to determine age). Preprocessing isn't a prominent part of the examples in this chapter, because this phase of data extraction has essentially already been done for you in the data set you'll be using here. Preprocessing plays a larger role in the examples in chapters 16 and 17.

14.2.1 Transforming raw data

Once you identify the features you want to try out, they must be converted into a format that's classifiable. This involves rearranging the data into a single location and transforming it into an appropriate and consistent form.

> **NOTE** Classifiable data consists of records with identical fields that contain one of four types of data: continuous, categorical, word-like, or text-like. Each record contains the fully denormalized description of one training example.

At first glance, it may seem like this step is already done: if the data looks like words, it must be words, right? If the data looks like numbers, it must be continuous, yes? But as you saw in chapter 13, first appearances can be a little misleading. For example, a ZIP code looks like a number but it's actually a category, a label for predetermined classes. Things that contain words may be word-like, or they may best be considered as categories or text. ID codes for users or products may look like numbers, categorical, or word-like data, but more commonly, they should be denormalized away in favor of the characteristics of the user or product that they link to.

The following example from computational marketing provides an exercise in preparing raw data for use in classification.

14.2.2 Computational marketing example

Suppose you're trying to build a classification model that will determine whether a user will buy a particular product if they're offered it. This isn't quite a recommendation system because it'll classify examples by using user and product characteristics rather than the collective behavior of similar users.

For this example, you have a database with several tables in it, as shown in figure 14.3. The tables are typical of a highly simplified retail system. There are tables representing users and products, a table recording when

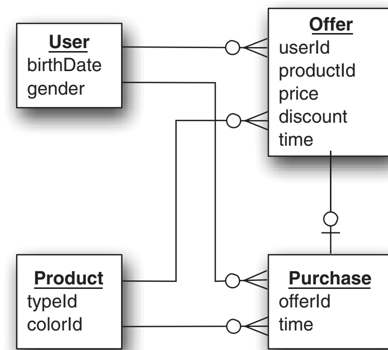


Figure 14.3 Table structure for a product sales example containing different types of data. Because no record in any table contains the data necessary in a training example for a classifier, this organization of raw data isn't directly usable as training data.

products were offered or shown to users, and a table that records purchases that arise from showing a product to a user. This data isn't currently acceptable as training or test data for a classifier because it's spread across several tables.

The marketing project depicted in figure 14.3 has a wide variety of data types. Users have demographic data, such as birth date and gender; products have type and color. Offers of products to users are recorded in the offer table, and purchases of products subsequent to an offer are recorded in the purchase table.

Figure 14.4 shows how this data might become training data for a classifier. There would be one of these records for each record in the offer table, but note how user and product IDs have been used to join against the user and product tables. In the process, the user's birth date is expressed as an age. An outer join has been used to derive the delay between offer and purchase as well as to add a flag indicating whether a purchase was ever made.

In order to get the data represented in the tables into the usable form shown in figure 14.4, it needs to be brought together and rearranged. To do this, you might use an SQL query like this:

```
select
  now()-birthDate as age, gender,
  typeId, colorId, price, discount, offerTime,
  ifnull(purchase.time, 0, purchase.time - offer.time) as purchaseDelay,
  ifnull(purchase.time, 0, 1) as purchased
from
  offer
  join user using (userId)
  join product using (productId)
  left outer join purchase using (offerId);
```

This query denormalizes the data stored in the separate database tables into records that contain all the necessary data. The driving table here is the offer table, and the nature of the foreign keys for `userId`, `productId`, and `offerId` force the output of this query to have exactly one record for each record in the original offer table. Note how the join against the purchase table is an outer join. This allows the `ifnull` expressions involving `purchase.time` to produce 0s if there is never a purchase.



Figure 14.4 To get classifiable data, training examples are constructed from several sources to denormalize everything into a single record that describes what happened. Some variables are transformed; birth date is expressed as an age, and purchase time is expressed as a delay.

NOTE Sometimes age is better for classification, and sometimes birth date is better. For instance, in the case of insurance data on car accidents, age will be a better variable to use because having car accidents is more related to life-stage than it is to the generation a person belongs to. On the other hand, in the case of music purchases, birth date might be more interesting because people often retain early music preferences as they get older. Their tastes often reflect those of their generation.

The records that come out of this query are now in the form of classifiable data, ready to be parsed and vectorized by the training algorithm. The next steps can be done using code you write or by putting the data into a format accepted by Mahout classifiers (which contain their own parsing and vectorization code). The following section and examples focus on vectorizing the parsed, classifiable data.

14.3 Converting classifiable data into vectors

In Mahout parlance, a `Vector` is a data type that stores floating-point numbers indexed by integers. This section of the chapter will show you how to encode data as `Vector`s, explain what feature hashing is, and demonstrate how the Mahout API does feature hashing. We also look at how you can encode the different types of values associated with variables.

You saw `Vector`s used previously in the chapters on clustering. Many kinds of classifiers, especially ones available in Mahout, are fundamentally based on linear algebra and therefore require training data to be input in the form of `Vector`s.

14.3.1 Representing data as a vector

How do you represent classifiable data as a `Vector`? There are several ways to do this, and they're summarized in table 14.1.

Table 14.1 Approaches to encoding classifiable data as a vector

Approach	Benefit	Cost	Where used?
Use one `Vector` cell per word, category, or continuous value	Involves no collisions and can be reversed easily	Requires two passes (one to assign cells and one to set values) and vectors may have different lengths	When dumping a Lucene index as `Vector`s for clustering
Represent `Vector`s implicitly as bags of words	Involves one pass and no collisions	Makes it difficult to use linear algebra primitives, difficult to represent continuous values, and you must format data in a special non-vector format	In naive Bayes
Use feature hashing	Involves one pass, vector size is fixed in advance, and it's a good fit for linear algebra primitives	Involves feature collisions, and the interpretation of the resulting model may be tricky	In `OnlineLogisticRegression` and other SGD learners

The approaches listed in table 14.1 are used in different classifiers in Mahout. Let's begin by looking at how to encode word-like, text-like, and categorical values as `Vector`s.

ONE CELL PER WORD

One way to encode classifiable data as a `Vector` involves making two passes through the training data: one to determine the necessary size of the `Vector` and build a dictionary that remembers where every feature should go in the `Vector`, and then another to convert the data. This approach can use a simple representation for encoding training and test examples: every continuous value and every unique word or category in categorical, text-like, and word-like data is assigned a unique location in the vector representation.

The obvious downside to this approach is that passing through the training data twice has the potential for making the training of the classifier twice as computationally expensive, which can be a real problem with very large data sets.

This two-pass approach is used by most of the clustering algorithms in Mahout.

VECTORS AS BAGS OF WORDS

Another approach is to carry around the feature names or names plus category, word, or text values instead of carrying around a `Vector` object. This method is used primarily by Mahout classifiers like naive Bayes and complementary naive Bayes.

The advantage of this approach is that it avoids the need for a dictionary, but it means that it's difficult to make use of Mahout's linear algebra capabilities that require known and consistent lengths for the `Vector` objects involved.

FEATURE HASHING

SGD-based classifiers avoid the need to predetermine vector size by simply picking a reasonable size and shoehorning the training data into vectors of that size. This approach is known as *feature hashing*. The shoehorning is done by picking one or more locations by using a hash of the name of the variable for continuous variables, or a hash of the variable name and the category name or word for categorical, text-like, or word-like data.

This hashed feature approach has the distinct advantage of requiring less memory and one less pass through the training data, but it can make it much harder to reverse-engineer vectors to determine which original feature mapped to a vector location. This is because multiple features may hash to the same location. With large vectors or with multiple locations per feature, this isn't a problem for accuracy, but it can make it hard to understand what a classifier is doing.

14.3.2 Feature hashing with Mahout APIs

In this section, we look at how to do feature hashing using the Mahout API. We present in detail how to encode continuous, categorical, word-like, and text-like features. We also explain what feature collisions are and how they may affect your classifier.

ENCODING CONTINUOUS FEATURES

Continuous values are the easiest kind of data to encode. The value of a continuous variable gets added directly to one or more locations that are allocated for the storage of the value. The location or locations are determined by the name of the feature.

Mahout supports hashed feature encoding of continuous values through `ContinuousValueEncoder`. By default, the `ContinuousValueEncoder` only updates one location in the vector, but you can specify how many locations are to be updated by using its `setProbes()` method. To encode a value for one variable, a `Vector` is needed: `new RandomAccessSparseVector(lmp.getNumFeatures())`. The value is then encoded with `encoder.addToVector(value, vector)`. Note that the value being encoded is assumed to be a string. If the value being encoded is available as a `double`, then pass a `null` for the `String` value and use the weight as the value: `encoder.addToVector(null, value, vector)`.

ENCODING CATEGORICAL AND WORD-LIKE FEATURES

Categorical features that have *n* distinct values can be encoded by storing the value 1 in one of *n* vector locations, to indicate which value the variable has. To decrease redundancy, it's also possible to use *n*–1 locations and then encode the *n*th category by storing a 1 in no location at all (all 0s).

Hashed feature encoding is similar, but the locations are scattered throughout the feature vector using the hash of the feature name and value. In addition, each category can be associated with several locations. An additional benefit of feature hashing is that the unknown and unbounded vocabularies typical of word-like variables aren't a problem.

To encode a categorical or word-like value using feature hashing, create a `WordValueEncoder`. This encoder is used in the same way the `ContinuousValueEncoder` was used, except that the default number of probes is two and the location being updated varies according to the value specified as well as the name of the variable. There are two kinds of this encoder: `AdaptiveWordValueEncoder` builds a dictionary on the fly to estimate word frequencies, so that it can apply larger weights to rare words; `StaticWordValueEncoder` uses a supplied dictionary or gives equal weights to all words if no dictionary is given.

Some learning algorithms are substantially helped by good word weighting, but others, like the `OnlineLogisticRegression` class described later, aren't much affected by equal weights for all words.

Encoding words with equal weighting is similar to how continuous values were encoded. Here is some sample code that shows how to encode continuous variables:

```
FeatureVectorEncoder encoder =
    new StaticWordValueEncoder("variable-name");

for (DataRecord ex: trainingData) {
  Vector v = new RandomAccessSparseVector(10000);
  String word = ex.get("variable-name");
  encoder.addToVector(word, v);
  // use vector
}
```

In this code, a `StaticWordValueEncoder` is constructed and assigned a name that determines the seed for a random number generator. Internally, values are encoded by combining the name of the variable and the word being encoded to get locations to be modified by the encoding operation.

Sizing the vector requires experimentation. Larger vectors consume more memory and slow down the training. Smaller vectors eventually cause so many feature collisions that the learning algorithm can't compensate.

ENCODING TEXT-LIKE FEATURES

Encoding a text-like feature is similar to encoding a word-like one except that text has many words. In fact, the example here simply adds up the vector representations for the words in the text to get a vector for the text. This approach can work reasonably well, but the more nuanced approach of counting the words first and then producing a representation that's a weighted sum of the vectors for unique words can produce better results. This second approach is illustrated later in this chapter.

> **TIP** Text-like data values are ordered sequences of words, but considering the ordering of the words in text is often not necessary to get good accuracy. Simply counting the words in a text, ignoring order, is usually sufficient. That being said, the best vector for text isn't usually just the sum of the vectors for the words in the text weighted by the counts of the words. Instead, the word vectors are usually best weighted by some function of the number of times that the corresponding word appeared in the text. Commonly used weight functions are the square root, the logarithm, or equal weighting for all words that appear, regardless of their frequency. The example in listing 14.1 uses linear weighting for simplicity, but weighting by the logarithm of the count is a better starting point for most applications.

Listing 14.1 shows how to encode text as a vector by encoding all of the words in the text independently, and then producing a linearly weighted sum of the encodings of each word. This is accomplished with a `StaticWordValueEncoder` and a way to break down or parse the text into words. Mahout provides the encoder and Lucene provides the parser.

Listing 14.1 Tokenizing and vectorizing text

```
FeatureVectorEncoder encoder = new StaticWordValueEncoder("text");
Analyzer analyzer =
    new StandardAnalyzer(Version.LUCENE_31);          <- Break text into words

StringReader in = new StringReader("text to magically vectorize");
TokenStream ts = analyzer.tokenStream("body", in);
TermAttribute termAtt = ts.addAttribute(TermAttribute.class);

Vector v1 = new RandomAccessSparseVector(100);        <- Encode into vector size 100
while (ts.incrementToken()) {
  char[] termBuffer = termAtt.termBuffer();
  int termLen = termAtt.termLength();
```

```
  String w = new String(termBuffer, 0, termLen);
  encoder.addToVector(w, 1, v1);
}
System.out.printf("%s\n", new SequentialAccessSparseVector(v1));
```

Add word w to vector v

This code produces a printable form of the vector, like this:

```
{8:1.0,21:1.0,67:1.0,77:1.0,87:1.0,88:1.0}
```

A graphical form of this output is shown in figure 14.5.

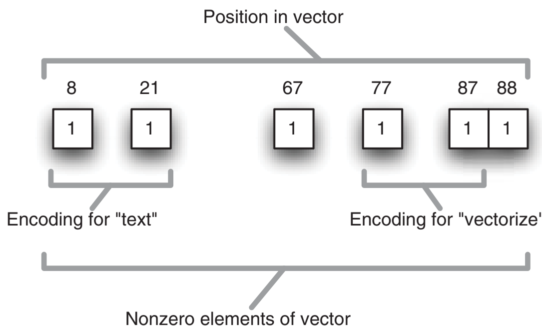


Figure 14.5 A vector that encodes *text to magically vectorize*. There are six nonzero values in this vector, which has a size of 100. Values equal to 0 aren't stored and thus aren't shown in results like these produced by the Lucene standard analyzer.

You can see in figure 14.5 that locations 8 and 21 encode the word *text*, locations 77 and 87 encode *vectorize*, and locations 67 and 88 encode *magically*. The word *to* is eliminated by the Lucene standard analyzer. The overall result of this process is a sparse vector, in which values equal to 0 aren't stored at all.

FEATURE COLLISIONS

The hashed feature representation for data can lead to collisions where different variables or words are stored in the same location. For instance, if you had used a vector with length 20 instead of the 100-long vector in the preceding example, you would have gotten the following result as the encoded value of *text to magically encode*:

```
{1:1.0,7:2.0,8:2.0,17:1.0}
```

In this vector, locations 7 and 8 have a value of 2 instead of 1, like all the nonzero cells in the vector shown in figure 14.5. How this happens is shown in figure 14.6.

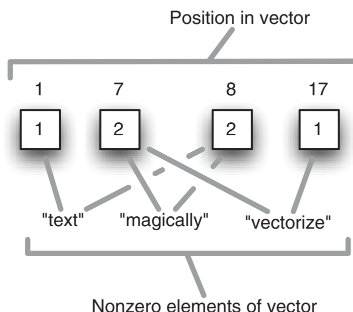


Figure 14.6 When the vector is size 20 instead of 100, collisions occur. The terms *text* and *magically* collide at location 8, and the terms *magically* and *vectorize* collide at location 7. These collisions make it more difficult to interpret the resulting vector, but typically they don't affect classification accuracy.

Dealing with feature collisions to avoid hurting performance can be fairly easy. In the vector shown in figure 14.6, the word *text* is encoded in locations 1 and 8, *magically* in locations 7 and 8, and *vectorize* in locations 7 and 17. This means that *text* and *magically* share location 8, and *vectorize* and *magically* share location 7. So locations 7 and 8 contain a value of 2, instead of 1 for all of the other nonzero values. Because each word has two locations assigned to it, the learning algorithm can learn to compensate for these collisions.

In this example, the learning algorithm can compensate for collisions because the transformation from words to vector is invertible. This would not normally be true if each word had a single location, because the vector dimension will almost always be smaller than the vocabulary of the text. With multiple locations, however, this vectorization technique usually works for the same reasons that Bloom filters work.

Where there is a collision between two words that have quite different meanings for the classifier, it's unlikely that the other locations for these two words will also collide and it's also unlikely that either will collide with a third word that presents the same problem. On average, therefore, this vectorization technique works well, and if the vector length chosen is large enough, there will be no measurable loss of accuracy. Exactly how long is long enough is subject to empirical validation by using the evaluation techniques you'll see in chapter 16.

Let's try these approaches with some real-world data using the SGD Mahout classification algorithm. (We discuss the different Mahout classification algorithms in section 14.5, and in section 14.6, you'll have another chance to classify the same data set with a different Mahout algorithm.)

14.4 Classifying the 20 newsgroups data set with SGD

In this section, you'll build a classification model for the 20 newsgroups data set using the learning algorithm known as SGD. In this example, you'll preview data and do the preliminary analysis, analyze which are the most common headers, convert the data to vector form through parsing and tokenizing, and train code for the 20 newsgroups data set project.

> **NOTE** The 20 newsgroups data set is a standard data set commonly used for machine learning research. The data is from transcripts of several months of postings made in 20 Usenet newsgroups from the early 1990s.

This example emphasizes feature extraction and focuses largely on the second phase of feature extraction, vectorization. The 20 newsgroups data set is useful here because the first phase of feature extraction, preprocessing raw data into classifiable data, will be relatively simple—the researchers who created the data set have already done most of the work.

14.4.1 Getting started: previewing the data set

The first step in preparing a data set is to examine the data and decide which features might be useful in classifying examples into categories of the chosen target variable—in this case each of the 20 newsgroups.

To begin, download the 20 newsgroups data set from this URL: http://people.csail.mit.edu/jrennie/20Newsgroups/20news-bydate.tar.gz. This version of the data set splits training and test data by date and keeps all of the header lines. The date split is good because it's more like the real world, where you'll want to process new data with your classifier rather than old data.

If you examine one of the files in the training data directory, such as 20news-bydate-train/sci.crypt/15524, you'll see something like this:

```
From: rdippold@qualcomm.com (Ron "Asbestos" Dippold)
Subject: Re: text of White House announcement and Q&As
Originator: rdippold@qualcom.qualcomm.com
Nntp-Posting-Host: qualcom.qualcomm.com
Organization: Qualcomm, Inc., San Diego, CA
Lines: 12

ted@nmsu.edu (Ted Dunning) writes:
>nobody seems to have noticed that the clipper chip *must* have been
>under development for considerably longer than the 3 months that
>clinton has been president. this is not something that choosing
...
```

The 20 newsgroups data set consists of messages, one per file. Each file begins with header lines that specify things such as who sent the message, how long it is, what kind of software was used, and the subject. A blank line follows, and then the message body follows as unformatted text.

The predictor features in this kind of data are either in the headers or in the message body. A natural step when first examining this kind of data is to count the number of times different header fields are used across all documents. This helps determine which ones are most common and thus are likely to affect our classification of lots of documents.

Because these documents have a simple form, a bash script like this can be used to count the different types of header lines:

```
#!/bin/bash
export LC_ALL='C'
for file in 20news-bydate-train/*/*
do
  sed -E -e '/^$/,$d' -e 's/:.*//' -e '/^[[:space:]]/d' $file
done | sort | uniq -c | sort -nr
```

This script scans all of the files in the training data set, deletes lines after the first blank line and deletes all content in the remaining lines after the colon. In addition, header continuation lines that start with a space are deleted. The header lines that remain in the output from `sed` are then sorted, counted, and sorted again in descending order by count.

The header line counts for the training examples are summarized in table 14.2. The row marked "..." indicates that we skipped to the end of the list to show you some of the more bizarre examples. The Action column shows what you might do with each header based on its potential usefulness. The choices are to discard the useless headers during parsing (Drop), retain potentially useful ones (Keep), or try some that are less clearly useful (Try?). It's worth using data whose value is unclear, as they may prove to be very helpful. Perhaps we should have marked them "Definitely try!"

Table 14.2 The most common headers found in the 20 newsgroups articles. A few of the less common headers are included as well.

Header	Count	Comment	Action
Subject	11,314	Text	Keep
From	11,314	Sender of message	Keep
Lines	11,311	Number of lines in message	Keep
Organization	10,841	Related to sender	Try?
Distribution	2,533	Possible target leak	Try?
Nntp-Posting-Host	2,453	Related to sender	Try?
NNTP-Posting-Host	2,311	Same as previous except for capitalization	Try?
Reply-To	1,720	Possible cue, not common, experiment with this	Try?
Keywords	926	Describes content	Keep
Article-I.D.	673	Too specific	Drop
X-Newsreader	588	Software used by sender	Drop
Summary	391	Like "Keywords"	Keep
Originator	291	Similar to "From," but much less common	Drop
In-Reply-To	219	Probably just an ID number, rare	Drop
News-Software	164	Software used by sender	Drop
Expires	113	Date-specific, rare	Drop
In-reply-to	101	Probably just an ID number, rare	Drop
To	80	Possible target leak, rare	Drop
X-Disclaimer	64	Noise, rare	Drop
Disclaimer	56	Possible cue due to newsgroup paranoia level, rare	Drop
...		...	
Weather	1	Bizarre header	Drop
Orginization	1	Spelling errors in headers are surprising	Drop
Oganization	1	Well, maybe not	Drop
Oanization	1		Drop
Moon-Phase	1	Yes, really, one posting had this; open source is wonderful	!?!

Many headers are unlikely to be helpful, and many are too rare to have much effect. Table 14.2 marks *Subject*, *From*, *Lines*, *Keywords*, and *Summary* as potentially interesting feature headers. These occur fairly often and seem likely to relate to the content of the document. The oddball among these is *Lines*, which is actually content-related because some newsgroups tend to have longer documents and some to have shorter ones.

> **TIP** Preliminary analysis of data is critical to successful classification. It's sometimes fun because the analysis often turns up Easter eggs like the *Moon-Phase* header line in table 14.2. These surprises can also be important in building a classifier, because they can uncover problems in the data or give you a key insight that simplifies the classification problem. Visualize early and visualize often.

Note that the 20 newsgroups example has been carefully prepared so it's easy to use for testing classifiers. As such, it isn't entirely realistic. You can see that no preprocessing is needed because all of the data is in one place and all obvious target leaks have already been removed, so you can proceed to text analysis.

14.4.2 Parsing and tokenizing features for the 20 newsgroups data

The second phase of preparing data for the learning algorithm is converting classifiable data (of the four possible types) into vectors. For the 20 newsgroups data set, all of the fields besides *Lines* are text-like or word-like, and all seem to be in a form that will tokenize easily using the standard Lucene tokenizer. The *Lines* field is a number, and it can be parsed using the Lucene tokenizer as well. It appears that this field will be quite helpful in distinguishing between some newsgroups because the average number of lines per message for the talk.politics.* groups is about 60 lines whereas the average number of lines for documents in misc.forsale is only 25 lines.

14.4.3 Training code for the 20 newsgroups data


No. 13

Now you can build a model. For simplicity in this example, we'll use the same `OnlineLogisticRegression` learning algorithm you used in the previous chapter for classifying dots. This time, however, you'll run the classifier from Java code instead of the command line, so that you can see how features are extracted and vectorized.

> **A look at logistic regression classifiers**
>
> Logistic regression classifiers use a linear combination of input values whose value is compressed to the (0, 1) range using the logistic function $1 / (1+e^{-x})$. The output of a logistic regression model can often be interpreted as probability estimates. In addition, the weights used in the linear combination can be efficiently computed in an incremental fashion, even when the feature vector has high dimension. This makes logistic regression a popular choice for scalable sequential learning. The features given to the logistic regression must be in numerical form, so text, words, and categorical values must be encoded in vector form.

SETTING UP VECTOR ENCODERS

First, you need objects that will help convert text and line counts into vector values, as follows:

```
Map<String, Set<Integer>> traceDictionary =
  new TreeMap<String, Set<Integer>>();
FeatureVectorEncoder encoder = new StaticWordValueEncoder("body");
encoder.setProbes(2);
encoder.setTraceDictionary(traceDictionary);
FeatureVectorEncoder bias = new ConstantValueEncoder("Intercept");
bias.setTraceDictionary(traceDictionary);
FeatureVectorEncoder lines = new ConstantValueEncoder("Lines");
lines.setTraceDictionary(traceDictionary);
Dictionary newsGroups = new Dictionary();
Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_31);
```

In this code, there are three different encoders for three different kinds of data. The first one, `encoder`, is used to encode the textual content in the subject and body of a posting. The second, `bias`, provides a constant offset that the model can use to encode the average frequency of each class. The third, `lines`, is used to encode the number of lines in a message.

Because the `OnlineLogisticRegression` class expects to get integer class IDs for the target variable during training, you'll also need a dictionary to convert the target variable (the newsgroup) to an integer, which is the `newsGroups` object.

CONFIGURING THE LEARNING ALGORITHM

You can configure the logistic regression learning algorithm as follows:

```
OnlineLogisticRegression learningAlgorithm =
    new OnlineLogisticRegression(
          20, FEATURES, new L1())
        .alpha(1).stepOffset(1000)
        .decayExponent(0.9)
        .lambda(3.0e-5)
        .learningRate(20);
```

The learning algorithm constructor accepts arguments specifying the number of categories in the target variable, the size of the feature vectors, and a regularizer. In addition, there are a number of configuration methods in the learning algorithm. The `alpha`, `decayExponent`, and `stepOffset` methods specify the rate and way that the learning rate decreases. The `lambda` method specifies the amount of regularization, and the `learningRate` method specifies the initial learning rate.

In a production model, the learning algorithm would need to be run hundreds or thousands of times to find good values. Here, we used a few quick trials to find reasonable values for these values.

ACCESSING DATA FILES

Next, you need a list of all of the training data files, as follows:

```
List<File> files = new ArrayList<File>();
for (File newsgroup : base.listFiles()) {
  newsGroups.intern(newsgroup.getName());
```

```
  files.addAll(Arrays.asList(newsgroup.listFiles()));
}

Collections.shuffle(files);
System.out.printf("%d training files\n", files.size());
```

This code puts all of the newsgroups' names into the dictionary. Predefining the contents of the dictionary like this ensures that the entries in the dictionary are in a stable and recognizable order. This helps when comparing results from one training run to another.

PREPARING TO TOKENIZE THE DATA

Most of the data in these files is textual, so you can use Lucene to tokenize it. Using Lucene is better than simply splitting on whitespace and punctuation because you can use Lucene's `StandardAnalyzer` class, which treats special tokens like email addresses correctly.

You'll also need several variables to accumulate averages for progress output such as average log likelihood, percent correct, document line count, and the number of documents processed:

```
double averageLL = 0.0;
double averageCorrect = 0.0;
double averageLineCount = 0.0;
int k = 0;
double step = 0.0;
int[] bumps = new int[]{1, 2, 5};
double lineCount;
```

These variables will allow you to measure the progress and performance of the learning algorithm.

READING AND TOKENIZING THE DATA

You're ready to process the data. You'll make only one pass through it because the online logistic regression algorithm learns quickly. Making only a single pass also facilitates progress evaluation, because each document can be tested against the current state classifier before it's used as training data.

To do the actual learning, documents are processed in a random order so that examples from different groups are mixed together. Presenting training data in a random order helps the `OnlineLogisticRegression` algorithm converge to a solution more quickly. You can do this as follows.

Listing 14.2 Parsing the data

```
for (File file : files) {
  BufferedReader reader = new BufferedReader(new FileReader(file));
  String ng = file.getParentFile().getName();                          <- 1 Identify newsgroup
  int actual = newsGroups.intern(ng);
  Multiset<String> words = ConcurrentHashMultiset.create();

  String line = reader.readLine();
  while (line != null && line.length() > 0) {                          2 Check for line count header
    if (line.startsWith("Lines:")) {                                   <-
```

```
      String count = Iterables.get(onColon.split(line), 1);
      try {
        lineCount = Integer.parseInt(count);
        averageLineCount += (lineCount - averageLineCount)
            / Math.min(k + 1, 1000);
      } catch (NumberFormatException e) {
        lineCount = averageLineCount;
      }
    }
    boolean countHeader = (
        line.startsWith("From:") || line.startsWith("Subject:")||
        line.startsWith("Keywords:")|| line.startsWith("Summary:"));
    do {
      StringReader in = new StringReader(line);
      if (countHeader) {
        countWords(analyzer, words, in);          ❸ Count words in headers
      }
      line = reader.readLine();
    } while (line.startsWith(" "));
  }
  countWords(analyzer, words, reader);            ❹ Count words in body
  reader.close();
}
```

The filename can be used to tell which newsgroup each document belongs to ❶. There is a header line that gives the number of lines, which can be encoded as a feature ❷. Then, in parsing header lines ❸ and the body of the document ❹, you need to count the words that you find. You can use a multi-set from the Google Guava library to do this counting. At least one of the documents has a bogus line count, and in that case it is reasonable to set the line count to the overall average to be defensive.

In each document, the headers come first and consist of lines containing a header name, a colon, content, and possible continuation lines that start with a space. The line number and the counts of the words on selected header lines are needed as features. After processing the headers, the body of the document is passed through the Lucene analyzer.

VECTORIZING THE DATA

With the data from the document in hand, you're ready to collect all of the features into a single feature vector that can be used by the classifier learning algorithm. Here's how to do that:

```
Vector v = new RandomAccessSparseVector(FEATURES);
bias.addToVector(null, 1, v);
lines.addToVector(null, lineCount / 30, v);
logLines.addToVector(null, Math.log(lineCount + 1), v);
for (String word : words.elementSet()) {
  encoder.addToVector(word, Math.log(1 + words.count(word)), v);
}
```

First, encode the `bias` (constant) term, which always has a value of `1`. This feature can be used by the learning algorithm as a threshold. Some problems can't be solved using logistic regression without such a term.

The line count is encoded both in raw form and as a logarithm. The division by 30 is done to put the line length into roughly the same range as other inputs so that learning will progress more quickly.

The body of the document is encoded similarly except that the weight applied to each word in the document is based on the log of the frequency of the word in the document, rather than the straight frequency. This is done because words occur more than once in a single document much more frequently than would be expected by the overall frequency of the word. The log of the frequency takes this into account.

MEASURING PROGRESS SO FAR

At this point, you should have a vector and be ready to give it to the learning algorithm. Until then, the document can still be considered to be withheld data. And as such, you might as well use the data to get some feedback about the (hopefully improving) accuracy of the classifier. You can compute two measures of accuracy: the log likelihood and the average rate of correct classification.

Log likelihood has a maximum value of 0, and random guessing with 20 alternatives should give a result of close to –3. Log likelihood gives credit even when the classifier gets the wrong answer if the right answer is near the top, and it takes credit away if it gets the right answer but has a wrong answer with nearly as high a score. These close-but-not-quite properties of log likelihood make it a useful measure for performance testing. But it can be difficult to explain log likelihood to non-technical colleagues, so it's good to have a simpler measure, the average rate of correct answers, as well.

You can compute the averages of the performance metrics using a trick known as Welford's algorithm, which allows you to always have an estimate of the current average. A variant on Welford's algorithm is used here, so that until you have processed 200 examples, a straight average is used. After that, exponential averaging is used so that you get a progressive measure of performance that eventually forgets about the early results from before the classifier had learned anything.

Here's a snippet that calculates the log likelihood and average percent correct:

```
double mu = Math.min(k + 1, 200);
double ll = learningAlgorithm.logLikelihood(actual, v);
averageLL = averageLL + (ll - averageLL) / mu;

Vector p = new DenseVector(20);
learningAlgorithm.classifyFull(p, v);
int estimated = p.maxValueIndex();

int correct = (estimated == actual? 1 : 0);
averageCorrect = averageCorrect + (correct - averageCorrect) / mu;
```

In this snippet, the model is asked for some performance metrics. The model computes the average log likelihood in the variable `averageLL`. Then the most highly rated newsgroup is determined in the `estimated` variable, and it's compared to the correct value. The result of the comparison is averaged to get the average percent correct as `averageCorrect`.

TRAINING THE SGD MODEL WITH THE ENCODED DATA

After getting progress information from the current training example, you can pass it to the learning algorithm to update the model. If the algorithm is making multiple passes through the data, it's probably a good idea to use testing examples for progress monitoring and training examples only for training, rather than using both types of examples for both training and progress monitoring, as we've done here.

In the example code, we used some fancy footwork to provide progress feedback at progressively longer intervals. This gives you fast feedback early in the run but doesn't flood you with data if you let the program run a long time. Here is how these progressively longer steps are computed:

```
learningAlgorithm.train(actual, v);
k++;
int bump = bumps[(int) Math.floor(step) % bumps.length];
int scale = (int) Math.pow(10, Math.floor(step / bumps.length));
if (k % (bump * scale) == 0) {
  step += 0.25;
  System.out.printf("%10d %10.3f %10.3f %10.2f %s %s\n",
       k, ll, averageLL, averageCorrect * 100, ng,
       newsGroups.values().get(estimated));
}
learningAlgorithm.close();
```

Each time the number of examples reaches `bump * scale`, another line of output is produced to show the current status of the learning algorithm. As the learning progresses, the frequency of status reports will decrease. This means that when accuracy is changing more quickly, there will be more frequent updates.

The last step is to tell the learning algorithm that it's done. This causes any delayed learning to take effect and cleans up any temporary structures.

14.5 Choosing an algorithm to train the classifier


No. 14

The main advantage of Mahout is its robust handling of extremely large and growing data sets. The algorithms in Mahout all share scalability, but they differ from each other in other characteristics, and these differences offer different advantages or drawbacks in different situations. Table 14.3 compares the algorithms used for classification within Mahout. This table, along with the rest of this section, will help you decide which Mahout algorithm bests suits a particular classification problem. Keep in mind that Mahout has additional algorithms not listed here—new ones continue to be developed.

Table 14.3 Characteristics of the Mahout learning algorithms used for classification

Size of data set	Mahout algorithm	Execution model	Characteristics
Small to medium (less than tens of millions of training examples)	Stochastic gradient descent (SGD) family: `OnlineLogisticRegression`, `CrossFoldLearner`, `AdaptiveLogisticRegression`	Sequential, online, incremental	Uses all types of predictor variables; sleek and efficient over the appropriate data range (up to millions of training examples)

Table 14.3 Characteristics of the Mahout learning algorithms used for classification *(continued)*

Size of data set	Mahout algorithm	Execution model	Characteristics
	Support vector machine (SVM)	Sequential	Experimental still; sleek and efficient over the appropriate data range
Medium to large (millions to hundreds of millions of training examples)	Naive Bayes	Parallel	Strongly prefers text-like data; medium to high overhead for training; effective and useful for data sets too large for SGD or SVM
	Complementary naive Bayes	Parallel	Somewhat more expensive to train than naive Bayes; effective and useful for data sets too large for SGD, but has similar limitations to naive Bayes
Small to medium (less than tens of millions of training examples)	Random forests	Parallel	Uses all types of predictor variables; high overhead for training; not widely used (yet); costly but offers complex and interesting classifications, handles nonlinear and conditional relationships in data better than other techniques

The algorithms differ somewhat in the overhead or cost of training, the size of the data set for which they're most efficient, and the complexity of analyses they can deliver.

14.5.1 Nonparallel but powerful: using SGD and SVM

As you saw in figures 13.1 and 13.7 (in chapter 13), the behavior of an algorithm can be powerfully scalable even if the algorithm is nonparallel. This section provides an overview of two sequential Mahout learning algorithms, stochastic gradient descent (SGD) and support vector machine (SVM).

THE SGD ALGORITHM

Stochastic gradient descent (SGD) is a widely used learning algorithm in which each training example is used to tweak the model slightly to give a more correct answer for that one example. This incremental approach is repeated over many training examples. With some special tricks to decide how much to nudge the model, the model accurately classifies new data after seeing only a modest number of examples. Although SGD algorithms are difficult to parallelize effectively, they're often so fast that for a wide variety of applications, parallel execution isn't necessary.

Importantly, because these algorithms do the same simple operation for each training example, they require a constant amount of memory. For this reason, each

training example requires roughly the same amount of work. These properties make SGD-based algorithms scalable in the sense that twice as much data takes only twice as long to process.

THE SVM ALGORITHM

An experimental sequential implementation of the support vector machine (SVM) algorithm has recently been added to Mahout. SVM implementations in Mahout include Java translations of the industrial-strength but nonscalable LIBLINEAR, which was previously only available in C++. The SVM implementation is still new and should be tested carefully before deploying.

The behavior of the SVM algorithm will likely be similar to SGD in that the implementation is sequential, and the training speed for large data sizes will probably be somewhat slower than SGD. The Mahout SVM implementation will likely share the input flexibility and linear scaling of SGD and thus will probably be a better choice than naive Bayes for moderate-scale projects.

14.5.2 The power of the naive classifier: using naive Bayes and complementary naive Bayes

The naive Bayes and complementary naive Bayes algorithms in Mahout are parallelized algorithms that can be applied to larger data sets than are practical with SGD-based algorithms, as indicated in table 14.3. Because they can work effectively on multiple machines at once, these algorithms will scale to much larger training data sets than will the SGD-based algorithms.

The Mahout implementation of naive Bayes, however, is restricted to classification based on a single text-like variable. For many problems, including typical large data problems, this requirement isn't a problem. But if continuous variables are needed and they can't be quantized into word-like objects that could be lumped in with other text data, it may not be possible to use the naive Bayes family of algorithms.

In addition, if the data has more than one categorical word-like or text-like variable, it's possible to concatenate your variables together, disambiguating them by prefixing them in an unambiguous way. This approach may lose important distinctions because the statistics of all the words and categories get lumped together. Most text classification problems, however, should work well with naive Bayes or complementary naive Bayes.

> **TIP** If you have more than ten million training examples and the predictor variable is a single, text-like value, naive Bayes or complementary naive Bayes may be your best choice of algorithm. For other types of variables, or if you have less training data, try SGD.

If the constraints of the Mahout naive Bayes algorithms fit your problem, they're good first choices. They're good candidates for data with more than 100,000 training examples, and they're likely to outperform sequential algorithms at ten million training examples.

14.5.3 Strength in elaborate structure: using random forests

Mahout has sequential and parallel implementations of Leo Breiman's random forests algorithm as well. This algorithm trains an enormous number of simple classifiers and uses a voting scheme to get a single result. The Mahout parallel implementation trains the many classifiers in the model in parallel.

This approach has somewhat unusual scaling properties. Because each small classifier is trained on some of the features of all of the training examples, the memory required on each node in the cluster will scale roughly in proportion to the square root of the number of training examples. This isn't quite as good as naive Bayes, where memory requirements are proportional to the number of unique words seen and thus are approximately proportional to logarithm of the number of training examples.

In return for this less desirable scaling property, random forests models have more power when it comes to problems that are difficult for logistic regression, SVM, or naive Bayes. Typically these problems require a model to use variable interactions and discretization to handle threshold effects in continuous variables. Simpler models can handle these effects with enough time and effort by developing variable transformations, but random forests can often deal with these problems without that effort.

Now that you're familiar with the learning algorithm options Mahout offers for training classifiers, it's time to put that knowledge to work. You've already used the SGD algorithm to train a classifier for the 20 newsgroups data in section 14.4; in the next section we use a different algorithm for the same problem, and you'll be able to compare the results of the two approaches.

14.6 Classifying the 20 newsgroups data with naive Bayes

The route to classification depends in part on the Mahout classification algorithm used. Whether the algorithm is parallel or sequential, as described in the preceding section, makes a big difference. The parallel SGD algorithm and the sequential naive Bayes algorithm have different input paths, and this difference requires different approaches to the handling of data, particularly with regard to vectorization.

This section will show you how to train a naive Bayes model on the same 20 newsgroups data set that you used in section 14.4. By using a different algorithm on the same data, you'll see the differences between sequential and parallel approaches to building a classification model.

We'll first go through the data extraction step needed to get the data ready for the training algorithm, and then you'll learn how to train the model. Once that's done, you'll be ready to begin the process of evaluating your initial model to determine whether it's performing well or changes need to be made.

14.6.1 Getting started: data extraction for naive Bayes

Let's get the data into a classifiable form and convert it to a file format for use with the naive Bayes algorithm. You can use the same 20 newsgroups data set you downloaded for the SGD example in section 14.4.

The naive Bayes classifiers can be driven programmatically, as you did with the SGD classifier, but it also has a parser built in that can be invoked from the command line. This parser accepts a variant on the data file format used by the SVMLight program, in which each data record consists of a single line in the file. Each line contains the value of the target variable followed by space-delimited features, where the presence of the feature name indicates that the feature has a value of 1 and absence indicates that the feature has a value of 0. In order to use the command-line version of the naive Bayes classifier, you'll have to reformat the data in the 20 newsgroups data set to be in this format.

The data in the 20 newsgroups data set has one directory per newsgroup, and inside each directory there's one document per file. You need to scan each directory and transform each file into a single line of text that starts with the directory name and then contains all the words of the document. The Mahout program called `prepare20newsgroups` does just this. To convert the training and test data, use a command like this:

```
$ bin/mahout prepare20newsgroups -p 20news-bydate-train/ \
       -o 20news-train/ \
       -a org.apache.lucene.analysis.standard.StandardAnalyzer \
       -c UTF-8

no HADOOP_CONF_DIR or HADOOP_HOME set, running locally
INFO: Program took 3713 ms

$ bin/mahout prepare20newsgroups -p 20news-bydate-test \
       -o 20news-test \
       -a org.apache.lucene.analysis.standard.StandardAnalyzer \
       -c UTF-8
no HADOOP_CONF_DIR or HADOOP_HOME set, running locally
INFO: Program took 2436 ms
```

In this command, the `-p` option specifies the name of the directory holding the training or test data. The `-o` option specifies the output directory. The `-a` option specifies the text analyzer to use, and the `-c` option specifies the character encoding that should be used to convert the input from bytes to text.

The result should be a directory called 20news-train that contains one file per newsgroup. Inside a data file like 20news-train/misc.forsale.txt, you'll see something like what is shown in figure 14.7.

Target variable   Text to classify

```
misc.forsale    from kedz@bigwpi.wpi.edu john kedziora subject ...
misc.forsale    from myoakam@cis.ohio-state.edu micah r yoakam subject ...
misc.forsale    from gt1706a@prism.gatech.edu maureen l eagle subject ...
misc.forsale    from mike diack mike-d@staff.tc.umn.edu subject make ...
misc.forsale    from jvinson@xsoft.xerox.com jeffrey vinson subject ...
misc.forsale    from hungjenc@usc.edu hung jen chen subject test ...
```

Figure 14.7 Converted training data in the format preferred by the naive Bayes program. Note that the lines displayed here have been shortened considerably.

14.6.2 Training the naive Bayes classifier

Up to this point, you've been working with the 20 newsgroups data, extracting features to prepare the data appropriately for input to the naive Bayes algorithm. With the training and test data converted to the right format, you can now train a classification model.

Try this command to produce a trained model using the naive Bayes algorithm:

```
bin/mahout trainclassifier -i 20news-train \
            -o 20news-model \
            -type cbayes \
            -ng 1 \
            -source hdfs
...
INFO: Program took 250104 ms
```

The result is a model stored in the 20news-model directory, as specified by the `-o` option. The `-ng` option indicates that individual words are to be considered instead of short sequences of words. The model consists of several files that contain the components of the model. These files are in a binary format and can't be easily inspected directly, but you can use them to classify the test data with the help of the `testclassifier` program.

You have now built and initially trained a model using the 20 newsgroups data and the naive Bayes algorithm. How well is it working? You're ready to address that question by beginning to evaluate the performance of your model.

14.6.3 Testing a naive Bayes model

It's now time to evaluate the performance of your newly trained model. We just introduce the evaluation process here, and you'll do some initial testing on your newly trained model. In chapter 15, we look at this topic in depth.

To run the naive Bayes model on the test data, you can use the following command:

```
bin/mahout testclassifier -d 20news-test \
            -m 20news-model \
            -type cbayes \
            -ng 1 \
            -source hdfs \
            -method sequential
```

Here the `-m` option specifies the directory that contains the model built in the previous step. The `-method` option specifies that the program should be run in a sequential mode rather than using Hadoop. For small data sets like this one, sequential operation is preferred. For larger data sets, parallel operation becomes necessary to keep the runtime reasonably small.

When the test program finishes, it produces output like what follows. The summary has raw counts of how many documents were classified correctly or incorrectly.

```
...
Summary
```

```
-------------------------------------------------------
Correctly Classified Instances          :       6398   84.9442%
Incorrectly Classified Instances        :       1134   15.0558%
Total Classified Instances              :       7532
=======================================================
```

We've omitted some noisy output and just displayed the summary section from the output. Here, the naive Bayes model is performing well, with a score of nearly 85 percent correct. The summary doesn't tell you anything about the errors that were made, other than the gross number. For more details about what errors were made, you can look at the confusion matrix in the next section of output, which appears something like this:

```
Confusion Matrix
-------------------------------------------------------
a b c d e f g h i j k l m n o p q r s t <--Classified as
388                                       | 397 a = rec.sport.baseball
  386                                     | 396 b = sci.crypt
    396                                   | 399 c = rec.sport.hockey
      347                                 | 364 d = talk.politics.guns
        377                               | 398 e = soc.religion.christian
  12      304           18                | 393 f = sci.electronics
            281     14  43        21      | 394 g = comp.os.ms-windows.misc
              313       22 16             | 390 h = misc.forsale
     26 69       83 41                    | 251 i = talk.religion.misc
        45        225 13              11  | 319 j = alt.atheism
                    334           32      | 395 k = comp.windows.x
                      367                 | 376 l = talk.politics.mideast
          19 23 15      307 13            | 392 m =comp.sys.ibm.pc.hardware
                       16 335             | 385 n = comp.sys.mac.hardware
                            371           | 394 o = sci.space
                              393         | 398 p = rec.motorcycles
                             12 364       | 396 q = rec.autos
            11      22            305     | 389 r = comp.graphics
      102                           160   | 310 s = talk.politics.misc
                                      362 | 396 t = sci.med
Default Category: unknown: 20
```

Your output will look a little different because we squashed the actual output here to fit on shorter lines to fit on the page. The confusion matrix shows a complete breakout of all correct and incorrect classifications so that you can see what kinds of errors the model made on the test data. Along the diagonal, notice that most newsgroups are classified fairly well, although talk.religion.misc and talk.politics.misc both stand out somewhat because of the relatively smaller number of correct classifications. About a third of the documents from talk.politics.misc are incorrectly classified as belonging to talk.politics.guns, which is at least plausible if not correct. Likewise, 69 documents from talk.religion.misc are classified as soc.religion.christian, which is, again, plausible. It's reassuring to see that the errors make a sort of sense, because that indicates that the model is picking up on the actual content of the documents in some meaningful way.

It's also ironically reassuring to see that the performance of the model isn't suspiciously good. For instance, if you run the same model on the same training data that it learned from, the summary looks like this:

```
bin/mahout testclassifier -d 20news-train -m 20news-model\
   -type cbayes -ng 1 -source hdfs -method sequential
...
Correctly Classified Instances          :      11075   97.8876%
Incorrectly Classified Instances        :        239    2.1124%
Total Classified Instances              :      11314
```

When given this familiar data, the model is able to get nearly 98 percent correct, which is much too good to be true on this particular problem. The best machine learning researchers only claim accuracies for their systems around 84–86 percent. We talk more about how to evaluate models in later chapters and, in particular, how to develop an idea about how well the model really should be performing.

14.7 Summary

In working through this chapter, you've gained experience with building and training classifiers using real data from the 20 newsgroups data set. The chapter focused particularly on how to do feature extraction and how to choose the best algorithm for your particular project.

Feature extraction is a big part of building a classification system, both in time, effort, and the payoff of doing it well. Remember that not all features are equally useful for every classifier, and that you'll need to try a variety of combinations to see what works. Raw data isn't usable directly. You saw in detail how to process raw data into classifiable form and how to convert that into the vectors required by Mahout's learning algorithms.

The types of values acceptable as input variables are different for the different learning algorithms, and section 14.5 provided an overview of those and other differences that distinguish one algorithm from another. You should now have a basic understanding of the benefits and costs of using the different algorithms to help guide you through making these choices in your own projects.

Finally, you saw the behavior of two different learning algorithms (SGD and naive Bayes) on the same data set, the 20 newsgroups.

Having learned a lot about the first stage of classification—training a model—you're now ready to move on to the second stage—evaluating and tuning the performance of a trained model. Those are the topics of the next chapter.

15 Evaluating and tuning a classifier

This chapter covers

- Basic considerations in evaluating classifiers
- Using the Mahout evaluation API
- Measuring the performance of an SGD classifier
- Common classifier problems and how to diagnose them
- Approaches for tuning a classifier

Because evaluation is so important, it's built into Mahout at a fundamental level. This chapter deals mainly with stage 2 of the classification process, the evaluation and fine-tuning of a classifier to prepare for deployment and to maintain performance in production. We discuss how classifiers are evaluated at a high level and the details of using the Mahout API for evaluation, and we look at an example of how to use the Mahout API. We also present several examples that highlight how performance metrics and diagnostic capabilities of the Mahout evaluation API can be used to diagnose common problems with classifiers. We finish this chapter with

a discussion of classifier-tuning strategies and techniques that cover the range from choosing an algorithm to adjusting learning rates.

Classifier evaluation presents several pitfalls, and this chapter also offers ways to avoid the more costly ones. Evaluation is also challenging because classification model internals can be difficult to understand.

15.1 Classifier evaluation in Mahout

Evaluation is an essential part of building a successful classifier. It involves far more than a single score of success after you've trained a model. Instead, it's an iterative process that begins during the training of a classifier. Early evaluation is valuable when building classifiers because it encourages you to try out many options, including different algorithms, settings, and sets of predictor variables. Evaluation also validates the data preprocessing pipeline for errors that could cause the classifier to appear to perform better than it actually will.

To evaluate classifiers, Mahout offers a variety of performance metrics. The main approaches are *percent correct*, *confusion matrix*, *AUC*, and *log likelihood*. The naive Bayes and complementary naive Bayes classifier algorithms are best evaluated using percent correct and confusion matrix. Any of these methods will work with the SGD algorithm; AUC or log likelihood may be particularly useful, because they provide insight into the model's confidence level.

This section covers how to get feedback on model performance, even during training, how to interpret the performance metrics that Mahout provides, and how to integrate cost considerations into your evaluation.

15.1.1 Getting rapid feedback

Evaluation in Mahout works a bit differently than in other systems. One useful difference is its ability to provide immediate and ongoing assessments of performance by allowing a user to pass target variable scores and reference values to an evaluator one at a time to get on-the-fly performance feedback. This Mahout approach is in contrast to other systems that must process all of the scores and target values in batch mode to evaluate performance. Mahout's ability to do online evaluation is convenient when you need to record the progress of a learning system, but it also has practical benefits in that it allows Mahout to adapt learning parameters on the fly as learning progresses.

The ability to get in-progress metrics is nice, but what do these metrics mean?

15.1.2 Deciding what "good" means

Intuitively, most of us want a classifier to be as accurate as possible, to put items into the correct category every time. Unfortunately, this intuition doesn't lead to a practical evaluation scheme. In the real world, no classifier is going to be 100 percent accurate. In fact, very high accuracy typically indicates something is wrong with the test—if it seems too good to be true, it probably is! So, how good is good? Useful evaluation requires realistic benchmarks for success so that you can recognize when a classifier is going to be effective.

Simple accuracy metrics such as percent correct aren't necessarily the best way to evaluate and tune a system. For example, consider two hypothetical models illustrated in table 15.1. A comparison of the two models in this table shows how a model that's "always wrong" based on a simple accuracy metric can be substantially more useful than a model that's sometimes right.

Table 15.1 Data from two hypothetical classifiers to show some of the limitations of just looking at percent correct. The columns show the frequency of each possible model output. Each row contains data for a particular correct value, and the answer with the highest score is in bold. Model 1 is never quite right, but may still be useful, whereas model 2 is like a stopped clock.

Correct Value	Model 1			Model 2		
	A	B	C	A	B	C
A	0.45	0.50	0.05	0.01	0.01	0.99
B	0.50	0.45	0.05	0.01	0.01	0.99
C	0.05	0.50	0.45	0.01	0.01	0.99

This hypothetical example shows why simply looking at the percent accuracy often doesn't reveal the true value of a model. In this table, each row represents an example with the correct answer on the far left. Each column contains the score for each of the three possible answers. Model 1 is never quite "right," yet it's much more useful than model 2 despite the higher apparent accuracy of the latter. Model 2 is scoring with a simplistic fixed rule and is correct part of the time purely at random. Model 1, in contrast, does appear to consistently eliminate one of three possibilities that isn't the right answer.

For many applications, model 1's output is probably better for many purposes than model 2's, which gets the right answer (sometimes) while never changing what it says. For this example, model 1 got 0 percent correct and model 2 got 25 percent correct. In contrast, model 1 has an average log likelihood of –0.8 whereas model 2 has the considerably worse value of –3.5. In the next section, we discuss measures of success like log likelihood that better reflect the types of performance we'd like to see.

Further, it's useful for a model to know when it's likely to be correct and when it doesn't have a good guess. There are performance measures that can reward this kind of self-knowledge. Model 1 in the previous example provides this information by showing it's nearly ambivalent about two choices, one of which is in fact right, whereas model 2 seems to have no clue about whether it's right or wrong. Again, log likelihood highlights this difference correctly.

Sometimes, however, a universal measure of performance isn't the right thing. This commonly happens when some errors are much worse than others.

15.1.3 Recognizing the difference in cost of errors

One issue that clouds the usefulness of simple accuracy metrics is that some errors cost more than others. The cost of a false positive may be much less than the cost of a false negative. For example, finding a cancer that doesn't exist has an expense in terms of retesting costs and anxiety, but not finding a cancer that does exist can cost a life.

A less dramatic example of the differential cost for errors of omission and commission is spam detection. The user of a spam detector is likely to grumble if spam is marked as non-spam, but a user will be furious if a non-spam message is marked as spam. Spam that is misclassified as non-spam wastes tens of seconds of the user's time for each error, but a non-spam message that's marked as spam is much more than an inconvenience. If this is frequent, the user likely will discover the problem and take action. If, however, such errors are rare, the users may not be aware of the need to compensate for their possibility, and the consequences will escalate.

The output of the Mahout classifier evaluation API was shown in the examples in chapters 13 and 14. The command-line tools we demonstrated in those chapters provided evaluation output. In some cases, those standard diagnostics suffice, but generally you'll need to use the API in your own programs to evaluate how things are going. The next section provides a detailed guide on how to use API-based evaluation in Mahout.

15.2 The classifier evaluation API

Mahout's classifier evaluation API is a collection of classes that compute various classifier performance metrics. These evaluation classes may be useful regardless of whether or not you use the Mahout classifiers.

The Mahout API classes that support various classifier metrics are shown in table 15.2. Each metric is described in more detail later in this section of the chapter. These topics include how to compute performance metrics in both online and offline applications. The online learning algorithms also provide API access to several of these performance metrics during training.

Table 15.2 Mahout supports a variety of classifier performance metrics through multiple APIs.

Metric	Supported by class
Percent correct	CrossFoldLearner
Confusion matrix	ConfusionMatrix, Auc
Entropy matrix	Auc
AUC	Auc, OnlineAuc, CrossFoldLearner, AdaptiveLogisticRegression
Log likelihood	CrossFoldLearner

Table 15.3 The Mahout classes that support performance evaluation for classifiers

Class	Description
`Auc`	
`OnlineAuc`	
`OnlineSummarizer`	
`ConfusionMatrix`	
`AbstractVectorClassifier`	`OnlineSummarizer`
`CrossFoldLearner`	`auc(), percentCorrect(), aloglikelihood()`
`AdaptiveLogisticRegression`	`CrossFoldLearnerauc()`

As you can see in table 15.2, not all metrics are supported uniformly. The capabilities are listed by class in table 15.3.

Note that the use of these classes varies. The learning algorithms, including `AdaptiveLogisticRegression`, `CrossFoldLearner`, and `AbstractVectorClassifier` all provide evaluation methods for the model being learned. In contrast, `Auc` and `OnlineAuc` are independent classes that compute performance metrics given scores and reference target values. Finally, the `OnlineSummarizer` computes aggregate statistics for arbitrary measurements.

So now let's take a look at how to compute these metrics in detail.

15.2.1 Computation of AUC

The AUC metric is useful when evaluating a model that produces a continuous score for a binary target variable. For example, a classifier that computes the probability between 0 and 1 that an item has a certain attribute or not fits this description. The score need not be a probability estimate, and models with very different score ranges can be compared using AUC.

You don't need to understand the meaning of AUC in depth. It's sufficient to remember that AUC is the probability that a randomly chosen *yes* instance will have a higher score than a randomly chosen *no* instance. A model that produces scores with no correlation to the target variable will produce an AUC of about 0.5, whereas a perfect model will produce an AUC of 1. An AUC in the range of 0.7 to 0.9 is typically considered good. For fraud detection or click-through analysis, this is the level at which a model produces usefully accurate results. AUC can have values as low as 0 in cases where the model is actually predicting the opposite of the target variable.


No. 15

Practical and theoretical considerations dictate that an AUC is best computed by holding out some subset of all data as test data, and comparing the classifier's result on the test data against the known value. Unfortunately, AUC is normally limited to models with binary target variables that produce a score rather than a hard classification. There are extensions to AUC to allow nonbinary target variables, but Mahout doesn't support these extensions yet.

In Mahout, if scores and gold-standard target values for binary test data are available, then there are a variety of ways to compute AUC. These include `Auc` from the `org.apache.mahout.classifier.evaluation` package. In addition, there are implementations of the `OnlineAuc` interface and `GlobalOnlineAuc` or `GroupedOnlineAuc` from the `org.apache.mahout.math.stats` package. With all of these classes, you need to feed the actual value of the target variable from withheld test data and the score for that example as computed by the classifier being evaluated. For all of these classes, the value of the AUC metric is obtained from the `auc()` method.

`OnlineAuc` differs from `Auc` in that `OnlineAuc` makes estimates of AUC available at any time during the loading of data with little overhead, though the insertion of each element is slightly more expensive. The algorithms used in `Auc` and `OnlineAuc` provide essentially identical accuracy, but `Auc.auc()` is more expensive to run than `OnlineAuc.auc()` because it involves sorting several thousand samples.

The following code listing shows how scores and target variable values can be read from a file to compute AUC using both classes.

Listing 15.1 Passing data for AUC metric classes

```
Auc x1 = new Auc();
OnlineAuc x2 = new GlobalOnlineAuc();
BufferedReader in = new BufferedReader(new FileReader(inputFile));
int lineCount = 0;
String line = in.readLine();
while (line != null) {
  lineCount++;
  String[] pieces = line.split(",");
  double score = Double.parseDouble(pieces[0]);
  int target = Integer.parseInt(pieces[1]);
  x1.add(target, score);
  x2.addSample(target, score);
  if (lineCount%500 == 0) {
    System.out.printf("%10d\t%10.3f\t%10d\t%.3f\n",
                lineCount, score, target, x2.auc());      ❶
  }
  line = in.readLine();
}
System.out.printf("%d lines read\n", lineCount);
System.out.printf("%10.2f = batch estimate\n", x1.auc())
System.out.printf("%10.2f = on-line estimate\n", x2.auc());   ❷
```

❶ x2 computes AUC as input is read

❷ xl and x2 results are essentially identical

In this code, `x1` and `x2` compute nearly the same estimate of AUC, but with `x2`, a progressive estimate of the value is available as the data is being scanned ❶. At the end, both `x1` and `x2` give essentially the same estimate by somewhat different methods ❷.

A recently computed value of AUC or log likelihood is available during training from the `auc()` and `getLogLikelihood()` methods of the `CrossFoldLearner` and `AdaptiveLogisticRegression` classes. Accessing these values allows you to compare different models as they're trained. The `AdaptiveLogisticRegression` class does this so that it can adapt the training parameters during training.

Although AUC is typically the gold standard for binary target variables and scored outputs, where there are more than two possible categories for the target variable, or where there are no scores, you need to look for another good metric.

15.2.2 Confusion matrices and entropy matrices

Probably the most straightforward metric for unscored outputs is the confusion matrix. A confusion matrix is a cross-tabulation of the output of the model versus the known correct target values in the data. The matrix has rows for each actual value of the target variable and columns for the output of the model. The cell in the row labeled *i* and column labeled *j* of the table is a count that tells you how often a test example from category *i* was thought by the model to be in category *j*. The confusion matrix for a good model will be dominated by counts along the diagonal. These diagonal elements correspond to cases where category *i* was correctly identified as category *i*.

Here is an example confusion matrix:

```
=======================================================
Confusion Matrix
-------------------------------------------------------
a    b    c    d    e    f    <--Classified as
9    0    1    0    0    0     |  10       a      = one
0    9    0    0    1    0     |  10       b      = two
0    0    10   0    0    0     |  10       c      = three
0    0    1    8    1    0     |  10       d      = four
1    1    0    0    7    1     |  10       e      = five
0    0    0    0    1    9     |  10       f      = six
Default Category: one: 6
```

The synthetic data used in this example is representative of an accurate classifier, as indicated by the fact that the numbers on the diagonal dominate the counts for each row.

Listing 15.2 shows how to use the `ConfusionMatrix` from the `org.apache.mahout.classifier` package to build a confusion matrix for data that includes the true category as found in the training or test data and the model score output. This code makes two passes through the data: once to get a list of all possible values of the target variable, and once to actually fill in the counts in the confusion matrix. It's common for the first pass to not be explicit because the list of all possible values is known by some other method. The critical point in this code is that you pass in the known correct value for the target variable and the value of the target variable as computed using the model being evaluated.

Listing 15.2 Building a confusion matrix

```
BufferedReader in = new
     BufferedReader(new FileReader(inputFile));
List<String> symbols = new ArrayList<String>();
String line = in.readLine();
while (line != null) {
  String[] pieces = line.split(",");
  if (!symbols.contains(pieces[0])) {
```

Reads and remembers values

```
    symbols.add(pieces[0]);
  }
  line = in.readLine();
}

ConfusionMatrix x2 = new ConfusionMatrix(symbols, "unknown");

in = new BufferedReader(new FileReader(inputFile));        <-- Counts the pairs
line = in.readLine();
while (line != null) {
  String[] pieces = line.split(",");                   <-- Input contains target
  String trueValue = pieces[0];                            and model output
  String estimatedValue = pieces[1];
  x2.addInstance(trueValue, estimatedValue);               <-- x2 gets true answer
  line = in.readLine();                                        and score
}
System.out.printf("%s\n\n", x2.toString());
```

The confusion matrices produced by `ConfusionMatrix` depend on the outputs of a classifier being compared to each other, or to a threshold in the binary case. Such a comparison forces the model to commit to a single answer. This approach loses information about how certain the model is of the answers it's giving, because only the final answers count. When probability scores are available, you can avoid this loss by using a variation of a confusion matrix known as an entropy matrix.

In an *entropy matrix*, the rows correspond to actual target values, and the columns correspond to model outputs just as with a confusion matrix. The difference is that the elements of the matrix are the averages of the log of the probability score for each true or estimated category combination. Using the average log in this way is strongly motivated by mathematical principles based on approximating a probability distribution optimally. This average log probability expresses how unusual a particular classifier result is by comparing the classifier's likelihood of classifying into a certain category against how common that category is overall. A good model will have small negative numbers along the diagonal and will have large negative numbers in the off-diagonal positions. This is analogous to the way that a confusion matrix has large counts on the diagonal and small counts off the diagonal.

The entropy matrix has a useful connection with log likelihood in that the weighted average of the diagonal entries of the entropy matrix is the log likelihood. Log likelihood is normally computed by taking the average of the log of the score for the target category. We'll discuss log likelihood in more depth in the next section.

The `Auc` class can be used to compute both confusion matrices and entropy matrices, but it currently only supports binary target variables. You can add data to the `Auc` object and print the confusion and entropy matrices like this:

```
Matrix entropy = x1.entropy();
for (int i = 0; i < entropy.rowSize(); i++) {
  for (int j = 0; j < entropy.columnSize(); j++) {
    System.out.printf("%10.2f ", entropy.get(i, j));
  }
  System.out.printf("\n");
}
```

Right now, the Auc class only works for binary target variables. Extending the Auc class to support more than just binary target variables is a feature that may appear in a future version of Mahout.

When you have a model that produces probability score outputs, the log of the score for the correct answer is a useful measure of how good the model is. If this value is averaged over a large number of withheld examples, it provides a useful single measure of model quality. This measure is known as the average log likelihood. This term is often abbreviated as log likelihood, at the risk of some confusion.

15.2.3 Computing average log likelihood

Log likelihood is a way of scoring a model based on how certain or uncertain the model is about the answers it produces. Log likelihood requires that a model produce a probability score as an output. A model gets credit if it gives very low probabilities to incorrect answers and a high probability to the correct answer. A particularly useful feature of log-likelihood evaluation is that a model is given partial credit for narrowing the set of possible answers even if the model doesn't settle on exactly the right answer.

Log likelihood has the desirable mathematical property that you can only maximize a log-likelihood score if your model exactly reproduces the actual probability distribution for the target variable. Log likelihood and AUC complement each other. Log likelihood can be used with multiple target categories, whereas AUC is limited to binary outputs. AUC, on the other hand, can accept any kind of score; log likelihood requires scores that reflect probability estimates. AUC has an absolute scale from 0 to 1 that can be used to compare different models, whereas the scale for log likelihood is open-ended.

The value of log likelihood for a single training example isn't particularly useful because it varies so much from example to example, but the average over a number of examples is useful. You can use an OnlineSummarizer to average the log-likelihood estimates from a classifier and summarize the estimates using median, mean, and upper and lower quartiles like this:

```
OnlineSummarizer summarizeLogLikelihood = new OnlineSummarizer();
for (int i = 0; i < 1000; i++) {
  Example x = Example.readExample();
  double ll = classifier.logLikelihood(x.target, x.features);
  summarizeLogLikelihood.add(ll);
}

System.out.printf(
    "Average log-likelihood = %.2f (%.2f, %.2f) (25%%-ile,75%%-ile)\n",
    summarizeLogLikelihood.getMean(),
    summarizeLogLikelihood.getQuartile(1),
    summarizeLogLikelihood.getQuartile(2));
```

In this code, examples are read and a classifier computes the log likelihood, which is then passed to an OnlineSummarizer. After the input has been read, this summary can

be queried using the `getMean()` and `getQuartile()` methods to get descriptive statistics for the distribution of log likelihood.

Log likelihood has a maximum value of 0 and no bound on how far negative it can go. For highly accurate classifiers, the value of average log likelihood should be close to the average percent correct for the classifier, multiplied by the number of target categories. For less accurate classifiers, especially those with many target categories, the average percent correct tends to go to 0, making it difficult to compare classifiers. The log likelihood, on the other hand, can still distinguish better from worse classifiers even when the classifiers being compared are rarely or even never exactly correct, which makes the average percent correct be 0.


No. 16

For internal developer audiences, log likelihood is probably a better measure for comparing models, but when presenting results to a less technically sophisticated audience, percent correct or even a confusion matrix is probably going to work better to convey your results.

Metrics like AUC and log likelihood give you a picture of which models perform better or worse overall, but they can't tell you why a model is working well or why it isn't. In order to know that, it's necessary to look inside a model.

15.2.4 Dissecting a model

Dissecting a model is a way to see which features make a large difference in the output of the model. Mahout can do this for classes that extend `AbstractVectorClassifier` by using a `ModelDissector` from the `org.apache.mahout.classifier.sgd` package.

Remember that features used as predictor variables have to be tokenized and vectorized in order to be used by the learning algorithm. These feature vectors are among the clues that need to be examined to understand why a model is working. As input, a `ModelDissector` takes a feature vector, a trace of how the feature vector was constructed, and a model. It then tweaks the feature vector and observes how the model output changes. By averaging over a number of examples, the effect of different features can be determined. When a summary is requested, the `ModelDissector` returns a list of those variables and values that exhibit the largest average effects.

The next code snippets show how you can use a `ModelDissector` to list the most important variables affecting the output of a model. The first step is to close the model so that all learning and regularization is finalized, and the second is to set up the `ModelDissector` and associated trace dictionary.

```
model.close();

Map<String, Set<Integer>> traceDictionary =
    new HashMap<String, Set<Integer>>();
ModelDissector md = new ModelDissector();

encoder.setTraceDictionary(traceDictionary);
```

Once the preliminaries are out of the way, you need to iterate over a number of examples. For each example, the trace needs to be cleared and the model dissector updated with the feature vector, the trace, and the model.

```
for (... lots of examples ...) {
  traceDictionary.clear();
  Vector v = encodeFeatureVector(example, actual, leakType);
  md.update(v, traceDictionary, model);
}
```

Finally, you can interrogate the model dissector to get a summary of the model weights that are packaged together with feature names.

```
List<ModelDissector.Weight> weights = md.summary(100);
for (ModelDissector.Weight w : weights) {
  System.out.printf("%s\t%.1f\n",
         w.getFeature(), w.getWeight());
}
```

Here the loop that iterates over the examples is pseudo code, but the rest of the code is taken from actual working code. The key factor here is that the trace dictionary is used during encoding to record hints about how to dissect the model later. You don't have to push many of your training examples into the encoder to get lots of good hints. Generally, if you have a few thousand or few tens of thousands of examples, you should be good to go.

The output of the `ModelDissector` is helpful in detecting what is going on in a misbehaving model. In a properly trained model, the most important variables and values should make sense to an expert in the problem domain. In a broken model, however, it's common for the most important variables to be clearly nonsensical. You'll see examples of correctly working and misbehaving models in the next few sections.

15.2.5 Performance of the SGD classifier with 20 newsgroups

In chapter 14 (section 14.4) you wrote a classifier using the SGD algorithm to categorize data from the 20 newsgroups data set. In this section, we extend that example by focusing on evaluating the performance of the classification system. Here we dissect the entire learning program, complete with metrics, so you can see how the pieces work together. In following sections, we inject typical bugs into this example so that you can see their effect on performance.

The following listing shows a program that trains an `AdaptiveLogisticRegression` model on the 20 newsgroups data. It's similar to the example for training an SGD model in section 14.4.

Listing 15.3 A complete program to train a model with progress diagnostics

```
private static final Analyzer analyzer =
    new StandardAnalyzer(Version.LUCENE_30);
private static final FeatureVectorEncoder encoder =
    new StaticWordValueEncoder("body");
private static final FeatureVectorEncoder bias =
    new ConstantValueEncoder("Intercept");

public static void main(String[] args) throws IOException {
  File base = new File(args[0]);
```

```
  int leakType = 0;
  if (args.length > 1) {
    leakType = Integer.parseInt(args[1]);
  }
  Dictionary newsGroups = new Dictionary();
  encoder.setProbes(2);

  AdaptiveLogisticRegression learningAlgorithm =
      new AdaptiveLogisticRegression(20, FEATURES, new L1());
  learningAlgorithm.setInterval(800);                          <-- 1 Set short generation time
  learningAlgorithm.setAveragingWindow(500);                   <-- 2 Set small window for averages

  List<File> files = Lists.newArrayList();
     File[] directories = base.listFiles();
         Arrays.sort(directories, Ordering.usingToString());
  for (File newsgroup : directories) {
    if (newsgroup.isDirectory()) {
      newsGroups.intern(newsgroup.getName());
      files.addAll(Arrays.asList(newsgroup.listFiles()));
    }
}
  Collections.shuffle(files);                                  <-- 3 Randomize order of training

  int k = 0;
  for (File file : files) {
    String ng = file.getParentFile().getName();
    int actual = newsGroups.intern(ng);

       Vector v = encodeFeatureVector(file, actual, leakType);
    learningAlgorithm.train(actual, v);
    k++;

    State<AdaptiveLogisticRegression.Wrapper, CrossFoldLearner> best =
            learningAlgorithm.getBest();

    if (best != null && k % 500) {
      CrossFoldLearner model =
         best.getPayload().getLearner();
      double averageCorrect = model.percentCorrect();
      double averageLL = model.logLikelihood();
      System.out.printf("%d\t%.3f\t%.2f\t%s\n",
          k, averageLL, averageCorrect * 100, leakLabels[leakType % 3]);
    }
  }
  dissect(newsGroups, learningAlgorithm, files);               <-- 4 Dissect final best model
}
```

This program starts with the allocation and configuration of the learning algorithm. Here we chose `AdaptiveLogisticRegression` for simplicity because it isn't necessary to worry much about its learning parameters. You might also use a `CrossFoldLearner` if you don't mind tweaking learning parameters. The code configures the generation time for the underlying evolutionary algorithm to be fairly short because the data set is relatively small and the algorithm will learn quickly ❶. Similarly, we set the averaging window for diagnostics to be relatively short so that we get quick indications of how the learning algorithm is doing ❷.

In preparation for training, we need a list of all documents in the training set. We also want to order them randomly to help the algorithm learn more effectively. In order to train the model, we iterate over the randomly ordered document list ❸. Training involves conversion to vector form and the actual training of the model. Once the evolutionary algorithm inside the `AdaptiveLogisticRegression` gets going, it will provide the best-known model from its pool. We can use this model to get the log likelihood and percent correct averaged over recent training examples.

When we complete training, we dissect the model ❹. The details of this dissection are shown in the following listing.

Listing 15.4 Model dissection for 20 newsgroups example

```
private static void dissect(Dictionary newsGroups,
           AdaptiveLogisticRegression learningAlgorithm,
           Iterable<File> files) throws IOException {
  CrossFoldLearner model =
         learningAlgorithm.getBest().getPayload().getLearner();
  model.close();

  ModelDissector md = new ModelDissector();
  Map<String, Set<Integer>> traceDictionary =
           Maps.newTreeMap();
  encoder.setTraceDictionary(traceDictionary);
  bias.setTraceDictionary(traceDictionary);

  for (File file : permute(files, rand).subList(0, 500)) {
    String ng = file.getParentFile().getName();
    int actual = newsGroups.intern(ng);

    traceDictionary.clear();
    Vector v = encodeFeatureVector(file, actual, leakType);
    md.update(v, traceDictionary, model);
  }

   List<String> ngNames = Lists.newArrayList(newsGroups.values());
  for (ModelDissector.Weight w : md.summary(100)) {
    System.out.printf("%s\t%.1f\t%s\t%s\n", w.getFeature(), w.getWeight(),
       ngNames.get(w.getMaxImpact() + 1));
  }
}
```

❶ Close model to finalize weights

❷ Provide trace dictionary

❸ Clear dictionary to avoid memory clog

In dissecting the model, we need to first make sure that the model's parameters have been finalized ❶. Closing an online model doesn't prevent further training, but it does make sure that we have a consistent state to dissect.

In dissection, we need to provide the feature encoding with a so-called trace dictionary ❷. This records which predictor variables and values were assigned to which locations in the feature vector. This slows down the encoding by quite a bit, so it isn't something to be done normally. By clearing the trace dictionary on each example, or every few examples, we can avoid letting the trace dictionary grow too large in memory ❸. After encoding each example and recording the details of the feature extraction, we pass the details to the model dissector.

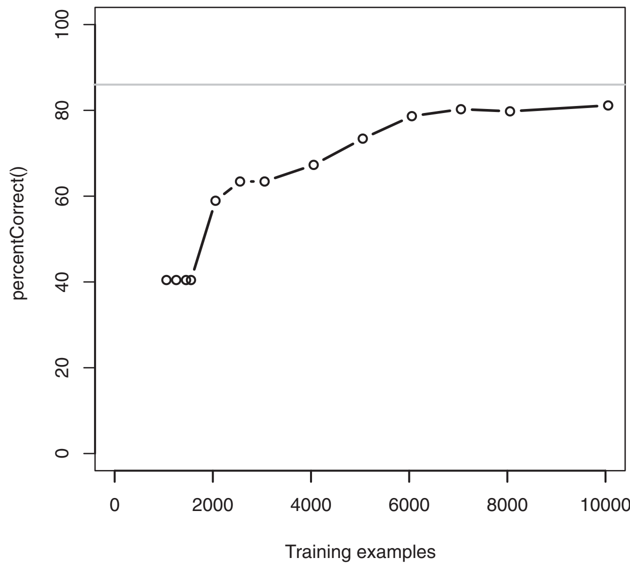


Figure 15.1 Increase in average percent correct with increasing number of training examples. The gray bar shows the reasonable maximum performance level that you can expect, based on the best results reported in the research literature.

Once you've finished processing a reasonable number of training examples, you can ask the model dissector for a list of the most important variables and values. In this example, the top 100 such values are printed.

If you run this example, you'll see the percent correct evolve something like what is shown in figure 15.1.

If you used multiple passes through the data used to create figure 15.1, the actual performance would likely approach state of the art performance in the mid 80 percent range. Making multiple passes doesn't improve the model as much as training on additional previously unseen data, because repeating the same data doesn't give as much diversity in training data. Happily, having too small a sample size is almost never the issue with Mahout-based classification, because Mahout's claim to fame is large data sets. Multiple passes through the data do still improve the model's performance because the system learns with every pass, up to a point.

Here is the output of the model dissection for a typical run such as shown in figure 15.1. The columns are the name of the feature (a single word of text, in this case), the largest weight of that feature, and the target variable value for that largest weight.

```
body=space       2.1   sci.space
body=sale        1.9   misc.forsale
body=car         1.9   rec.autos
body=windows     1.8   comp.os.ms-windows.misc
body=mac         1.7   comp.sys.mac.hardware
body=bike        1.7   rec.motorcycles
body=apple       1.5   comp.sys.mac.hardware
body=gun         1.5   talk.politics.guns
body=baseball    1.5   rec.sport.baseball
body=graphics    1.5   comp.graphics
body=key         1.5   sci.crypt
body=x           1.4   comp.windows.x
```

```
body=dod          1.4   rec.motorcycles
body=orbit        1.4   sci.space
body=clipper      1.4   sci.crypt
body=encryption   1.3   sci.crypt
body=window       1.3   comp.windows.x
body=image        1.2   comp.graphics
body=hockey       1.2   rec.sport.hockey
body=atheists     1.2   alt.atheism
```

These features are clearly pretty reasonable, with the possible exception of the word *dod*. This is apparently a feature for the `rec.motorcycles` Usenet discussion group. Examination of the training texts, interestingly, shows that there was a lengthy discussion back and forth using *dod* as a nonsense word. Although this feature is probably not going to help future performance after the presumably short-lived use of that word, it also isn't likely to hurt.

At this point, you know how to measure how good your classifier is. That leaves you with the problem of trying to decide what those measurements mean. Let's look at how typical problems show up in the performance metrics, and how you can diagnose and fix these problems.

15.3 When classifiers go bad

When working with real data and real classifiers it's almost a rule that the first attempts to build models will fail, occasionally spectacularly. Unlike normal software engineering, the failures of models aren't usually as dramatic as a null pointer dereference or out-of-memory exception. Instead, a failing model can appear to produce miraculously accurate results. Such a model can also produce results so wrong that it seems the model is trying to be incorrect. It's important to be somewhat dubious of extremely good or bad results, especially if they occur early in a model's development.

This section will help you understand how the metrics you collect, using techniques from the previous section, can tell you about these predictable bumps in the life of a classifier. To do that, you need to know what the common problems with classifiers are and how those problems show up in the performance metrics.

The two most common problems in building classification models are so-called target leaks and broken feature extraction. A *target leak* occurs when some feature in the training data provides information about the target variable in some way that won't happen in production. This is the classifier equivalent of handing out the answers to an exam along with the exam itself. Not surprisingly, a classifier can do quite well in such a case. This sort of problem can happen in subtle ways that are difficult to diagnose. *Bad feature extraction* can also be difficult to detect unless it's catastrophically bad. You'll see examples of both problems in the next sections.

15.3.1 Target leaks

A target leak is a bug that results from information from the target variable being included in the predictor variables used to train a classifier. The major symptom of a target leak is performance that's just too good to be true.

To help you understand how to avoid a target leak, we intentionally insert one in the 20 newsgroups sample classifier from listing 15.3. Here is the small addition to the program that injects a target leak into the training examples:

```
private static final SimpleDateFormat("MMM-yyyy");
// 1997-01-15 00:01:00 GMT
private static final long DATE_REFERENCE = 853286460;

...
  long date = (long) (1000 *
      (DATE_REFERENCE + target * MONTH + 1 * WEEK * rand.nextDouble()));
  Reader dateString = new StringReader(df.format(new Date(date)));
  countWords(analyzer, words, dateString);
```

Unlike before, this code injects a simulated target leak into the data in the form of a date field. This date field is chosen so that all the documents from the same newsgroup appear to have come from the same month, but documents from different newsgroups come from different months. Dates, times, and ID numbers are common sources of target leaks, but we could have used any sort of field to inject the leak. Even the target variable itself can accidentally be included as a predictor. As you might expect, such an error creates the mother of all target leaks.

When you train the model using this target leak in addition to the text, the accuracy of the model gets entirely too good. Figure 15.2 shows what happens. Especially in the case where the target leak appears with just the subject line, accuracy approaches 100 percent. This is unrealistically high, because the best reported result for this data set in the research literature is about 86 percent. In the case where the leak is added to all of the text from the subject line and the body of the message, it takes longer for the learning algorithm to sort out what is happening, but before long it still reaches implausibly good performance.

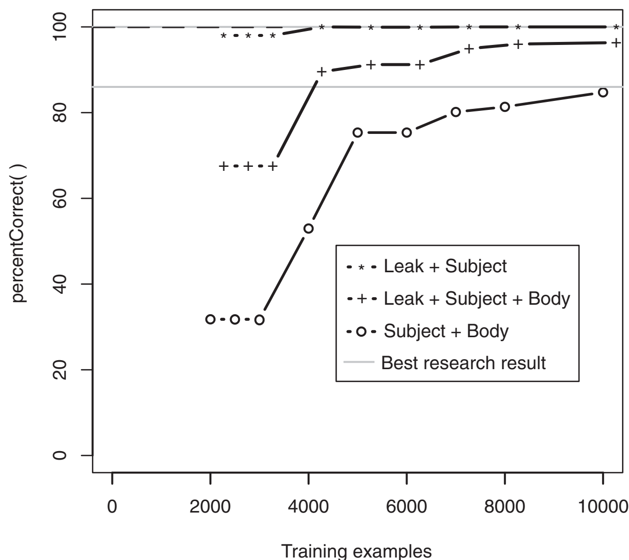


Figure 15.2 A target leak produces implausible accuracy, but not always instantly.

From the results shown in figure 15.2, you can see that the two lines representing conditions with the target leak quickly approach 100 percent accuracy. Such an extreme level of accuracy isn't plausible for this problem. The SGD model without the leak, on the other hand, only just reaches the level represented by the best academic research results for this problem. That level of performance is very good, but still plausible.

When working with real data rather than carefully studied research data, it isn't necessarily possible to known how good "too good" is from the start. If the model appears to be doing exceptionally well, or if the performance on new data is dramatically worse than predicted by cross validation, it's worth looking at the output of the `ModelDissector`.

Here are the first few lines of the model dissection output from one of the cases with a target leak. The columns in the output are the term, the weight, and the newsgroup that the weight applies to.

```
body=feb-1998          3.7   sci.electronics
body=dec-1997          3.6   comp.graphics
body=sep-1997          3.5   sci.med
body=mar-1998          3.3   comp.sys.mac.hardware
body=jul-1997          3.3   talk.religion.misc
body=jun-1997          3.2   misc.forsale
body=apr-1998          3.2   rec.motorcycles
body=jul-1998          3.1   talk.politics.misc
body=jun-1998          3.0   comp.os.ms-windows.misc
body=mar-1997          2.9   sci.space
body=apr-1997          2.9   talk.politics.guns
body=jan-1998          2.8   soc.religion.christian
body=nov-1997          2.8   comp.sys.ibm.pc.hardware
body=may-1997          2.8   comp.windows.x
body=feb-1997          2.7   rec.sport.baseball
body=aug-1998          2.6   alt.atheism
body=may-1998          2.6   rec.autos
body=oct-1997          2.5   rec.sport.hockey
body=gun               2.1   talk.politics.guns
body=aug-1997          2.1   sci.crypt
body=windows           2.0   comp.os.ms-windows.misc
body=consciously       1.8   sci.electronics
body=taber             1.8   sci.electronics
body=uw.pc.general     1.8   sci.electronics
body=hearing           1.8   sci.med
body=car               1.7   rec.autos
body=sdennis@osf.org   1.7   misc.forsale
body=1024x768x24       1.7   comp.graphics
body=contradicts       1.7   talk.religion.misc
body=market            1.7   comp.graphics
body=passes            1.7   misc.forsale
body=att-14,796        1.7   talk.religion.misc
body=bars              1.7   sci.med
```

These most heavily weighted features are dominated by dates. Other than large-scale seasonal changes, like Christmas, having the date be such a strong predictor of topic

isn't reasonable. Having nearly all of the top 20 features be particular dates is entirely too much to believe, so the only reasonable conclusion is that these features represent a target leak. Of course, in this case, we intentionally injected the target leak, so nobody should be surprised to find one.

Another, more subtle, indicator that something is wrong comes from looking at the top nonleak features. The `AdaptiveLogisticRegression` adjusts the learning rate in order to optimize performance. In the case of a target leak, it's to the advantage of the algorithm to decrease the training rate fairly quickly once it has found the leak features. As such, it can freeze other extraneous features without their weights decaying to 0, as a normal model would have to do to get good performance. As a result, you'll see words like *taber*, *market*, *passes*, and *bars* listed as highly weighted features, predictive of `sci.electronics`, `comp.graphics`, `misc.forsale`, and `sci.med` respectively. That such features persist in the final model is an indication that learning has frozen without proper pruning.

15.3.2 Broken feature extraction

Another common problem occurs when the feature extraction is somehow broken. In contrast to target leaks, where performance is unreasonably good, broken feature extraction causes much worse performance than expected.

The following code shows how a Lucene tokenizer should be used for the tokenization of the 20 newsgroups data.

```
TokenStream ts = analyzer.tokenStream("text", in);
ts.addAttribute(TermAttribute.class);
while (ts.incrementToken()) {
  String s = ts.getAttribute(TermAttribute.class).term();
  words.add(s);
}
```

An easy error to make would be to try to get the text token by using `s = ts.toString()` instead of using the correct, but slightly obscure, call to `term()`. Committing this error in tokenization causes additional information to be retained in addition to the desired token, and this additional information makes the same word in different locations appear as if it were two different words. That makes learning from the affected predictor hard, if not impossible.

Figure 15.3 shows what performance looks like with the tokenization error in place. The percent correct never gets much above 5 percent, which is, of course, the result the model would get by randomly selecting a result.

This example is extreme because all the input data has been corrupted by bad tokenization; nothing is left that the model can use to produce good results. In practice, it's more common that only some fields are corrupted and others are fine. Such a partial corruption will lead to degraded performance if the corrupted variable would have helped the model, but it won't necessarily lead to the flat-line performance that you see in this example because the uncorrupted variables are still available to carry some of the load.

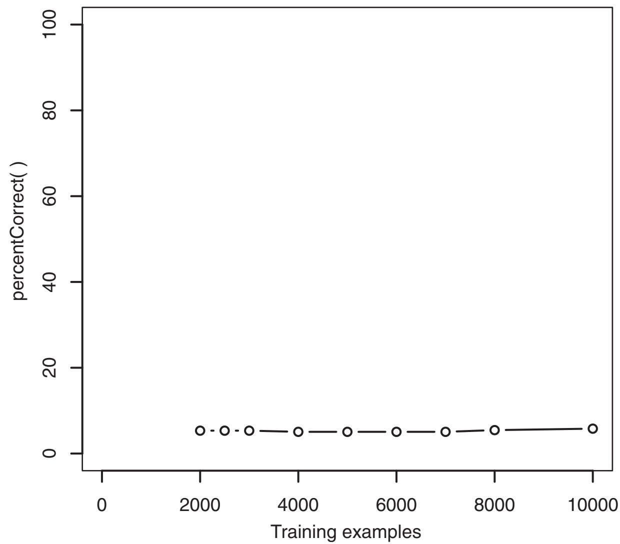


Figure 15.3 Performance with the tokenization bug never increases above the level expected from a random selection of the target variable. The gray line shows the performance that would be expected by chance.

The best way to find these kinds of problems is to compute summary statistics on the values being encoded. If the value is continuous, use an `OnlineSummarizer` to examine the mean, minimum, maximum, and quartiles. These values should seem reasonable. For categorical, word-like, and text-like data, you should produce counts of the tokens produced and plot these counts versus rank on a log-log scale. For the 20 newsgroups data, this gives a graph like the one shown in figure 15.4. For word-like and text-like data, something similar should emerge for each field of your data. In the case of our injected tokenization error, the frequencies of essentially all terms would be exactly 1.

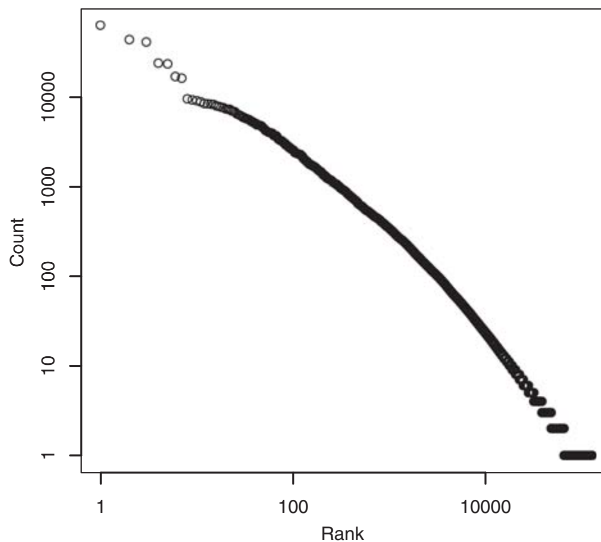


Figure 15.4 Plot of token count versus rank for a correctly operating tokenizer on the 20 newsgroups data. With the broken tokenizer, the counts would all have a value of 1.

Table 15.4 shows some of the incorrect tokens that are produced if `insertTokenError` is set along with the correct values that should have been produced. Keep in mind that the parentheses, commas, and numbers are all part of the incorrect tokens.

Table 15.4 How bad tokens can defeat feature extraction

Erroneous token	Correct token
(term=elastic,startOffset=1705,endOffset=1712,...)	elastic
(term=elastic,startOffset=2064,endOffset=2071,...)	elastic
(term=failed,startOffset=1236,endOffset=1242,...)	failed
(term=failed,startOffset=245,endOffset=251,...)	failed
(term=failed,startOffset=974,endOffset=980,...)	failed

Note how repeated instances of the same token in the table appear different if too much information is inadvertently retained. This situation makes these erroneous tokens useless for categorization because the chances that they'll appear in another document at exactly the same place are nearly nil.

Now that you know how some pitfalls such as target leaks and broken feature extraction can defeat a classifier, let's turn to a more optimistic topic and see how you can use evaluation to adjust and tune a classifier to improve performance.

15.4 Tuning for better performance

Beyond correcting errors in training data, you can do a lot to improve classifier performance. Some approaches involve making the classifier's life easier by changing the problem itself. Others involve making the classifier run better on the data given. In the first case, it's possible to find new predictor variables, combinations of predictors, or transformations of predictors to help the classifier out. It's also possible to eliminate predictors that aren't helping or to simplify the target variable. In the second case, learning rates can be tuned or feature encoding strategies adjusted in the learning algorithm.

In any case, when tuning a classifier system, you need to make careful and realistic measurements of performance with and without the changes. It isn't usually sufficient to examine a few ad hoc examples unless your change has made a dramatic difference. In fact, once the system is working reasonably, most adjustments should improve or degrade performance by only small amounts, which can only be detected with accurate measurement.

15.4.1 Tuning the problem

To most people, the idea of tuning up a classification system brings to mind the idea of tuning the learning algorithm so that its accuracy is as high as possible. This assumption leads to trouble, however, because it misses some of the best opportuni-

ties. Often the big improvements result from tuning the problem instead. This approach can include changing the input, the output, or even the interpretation of the output.

REMOVE FLUFF VARIABLES

Some variables don't help the classifier much or even at all, and it may be better to simply withhold these variables from the classifier. Paradoxically, removing some information like this can improve accuracy because the learning algorithm may have less distraction with fewer variables and thus may learn a better classifier in the available training time. The target leak problem shown in section 15.4 was a great example of this phenomenon. In that example, the learning algorithm took considerably more time to learn the trivial target leak model when it had lots of text around to distract it. Conversely, you could say that the leak variable itself distracted the learning algorithm and prevented it from noticing the really helpful variables in the subject or body.

Regularization is a more general version of this idea. Regularization is a mathematical technique that's used to avoid over-fitting by decreasing the effect of some variables or even eliminating them entirely. Regularization operates by having the learning algorithm learn to minimize error and model complexity at the same time. The mathematical function that penalizes complexity is called the *prior* in Mahout terminology.

Algorithms such as SGD with regularization try to eliminate variables, where possible, by penalizing the inclusion of variables in the model. All else equal, these algorithms would prefer to not include a variable. Even so, it can take time for these algorithms to understand which variables can be eliminated. If dissecting a model shows that some variables are never mentioned in the model, it's probably useful to try eliminating them so that subsequent training goes faster. Likewise, it's useful to test versions of the model that aren't allowed to use certain categories of variables, to see what happens.

Other Mahout classifier algorithms, like naive Bayes and complementary naive Bayes, can benefit greatly from the elimination of variables that don't carry useful information, because these algorithms have no mechanism to eliminate these variables on their own via regularization.

It's relatively common to find that removing one of two variables makes no difference, but removing both seriously hurts performance. It can also happen that using a particular variable alone gives no better performance than chance decisions. It would be tempting to eliminate that variable if you noticed this lack of performance. Occasionally, however, such a variable would be able to improve the performance of other variables, even though it shows no predictive qualities in isolation. It's safe to say that any variable that doesn't appear in the dissection of the final model can be eliminated with positive effect, but most other conclusions require careful testing.

ADD NEW VARIABLES, INTERACTIONS, AND DERIVED VALUES

In building a classification system, some data is easier to wrangle into usable form than other data. Difficulties in data preparation will usually lead to delays in being

able to present all of the desired predictor variables to the learning algorithm. Conversely, some variables will probably be available sooner than others. In a large project, the delays in getting certain predictors can easily extend to weeks or months, and some predictors may well turn out to be impossible to get.

The potential for these delays means that modeling and evaluation should be started early, with whatever data is available. It often happens that the initially available predictor variables are sufficient to build a deployable model. Additional variables should be tested as they become available, because they may improve performance even if the early models are usable in production.

Another common source of new variables is the transformation of existing continuous variables. It can pay to insert additional versions that are the logarithm or the square of the original value. Generalized linear models, such as the Mahout SGD-based models, can benefit from this technique, because they otherwise notice only linear relationships between predictors and categories. Most tree-based models won't notice the difference.

Here's an example where a `TrainingExample` object is queried for a line count that's then encoded three different ways:

```
FeatureValueEncoder lineEncoder =
    new ConstantValueEncoder("lines");
FeatureValueEncoder logLineEncoder =
    new ConstantValueEncoder("log(lines)");
FeatureValueEncoder lineSquaredEncoder =
    new ConstantValueEncoder("lines^2");

for (TrainingExample example : trainingData) {
  Vector v = new RandomAccessSparseVector(50000);

  double lines = example.getLines();
  lineEncoder.addToVector((byte[]) null, lines, v);
  logLineEncoder.addToVector((byte[]) null, Math.log(lines), v);
  lineSquaredEncoder.addToVector((byte[]) null, lines * lines, v);
}
```

Note name changes

By encoding the line count transformed in multiple ways, we've provided the model with the ability to make certain kinds of inferences much more easily. For example, including logs of original and sale prices gives the model access to the equivalent of a discount percentage. Giving it only the original and sale prices untransformed allows it to only see discounts in dollar terms rather than as a fraction of the original price. Which view is better is hard to say in advance, so letting the learning algorithm choose can be a very good idea.

It doesn't make sense to add multiple versions of a variable for a naive Bayes model because naive Bayes only accepts categorical or text-like inputs. These inputs are encoded as a number of binary values, and transformations of binary variables rarely make any useful sense.

Interaction variables are combinations of other variables, and they're another source of power for a classifier. For instance, in a marketing application, it might be very

good to have an interaction variable that combines gender with brand name. Without this interaction variable, the model can deduce that women are more or less likely to respond to a marketing offer than men, or that a particular brand is more or less popular. With the interaction of gender and brand, however, the model can say that a particular brand is more or less popular with men or more or less popular with women. Because different manufacturers specialize the brands they sell to appeal to different market segments, it's no surprise that such a variable can be very useful. With hashed feature representations, interaction variables can be added using code similar to the following:

```
LuceneTextValueEncoder authorEnc = new LuceneTextValueEncoder("author");
LuceneTextValueEncoder subjectEnc = new LuceneTextValueEncoder("subject");
FeatureVectorEncoder authorPlusSubjectEnc =
      new InteractionValueEncoder("author_subject", author, subject);

for (TrainingExample example : trainingData) {
  Vector v = new RandomAccessSparseVector(50000);
  String author = example.getAuthor();
  String subject = example.getSubject();
  authorEnc.addToVector(author, 1, v);
  subjectEnc.addToVector(subject, 1, v);
  authorPlusSubjectEnc.addToVector(author, subject, 1, v);
  ...
}
```

Here, the `authorPlusSubjectEnc` interaction encoder is created using a reference to the `authorEnc` author encoder and the `subjectEnc` subject text encoder. The string containing the author name and subject line is encoded by encoding author, subject, and the combination as an interaction variable. The interaction variable represents not just the author and subject, but the fact of their association, distinct from that subject as used by another author or that author talking about a different subject. This allows the classifier to draw different distinctions based on both who and what was said. If different authors use the same text to mean different things, the model would have no chance of learning the correct meaning without this interaction variable. But with it, the model can ascribe different meanings to exactly the same words, depending on the context in which they appear. In addition, because of the inclusion of the separate encoders for author and subject, the model can also learn subject line meanings that don't depend on who authored the subject line.

Interactions are powerful, but they can dramatically increase the complexity of a classifier and make feature encoding and learning dramatically slower. Because the number of features for each training example is dramatically increased, this can also cause accuracy to degrade.

Another way to introduce transformations of existing variables is to cluster a portion of training data, applying something like k-means clustering to several of the predictor variables. This clustering leads to a new predictor: for each piece of data, its cluster is a categorical feature.

Similarly, it's possible to build a classifier model and then use the output of that classifier as a predictor variable for another model. The idea is that the first model will learn the general outline of the problem, and the cascaded second model can learn how to correct errors that the first model makes. Here is a bit of code that illustrates this:

```
AbstractVectorClassifier subModel =
    ModelSerializer.readBinary(new FileReader("sub-model.model"),
    OnlineLogisticRegression.class);
ConstantValueEncoder subModelEnc = new ConstantValueEncoder("sub");
...
for (TrainingExample example : trainingData) {
  Vector v1 = new RandomAccessSparseVector(50000);
  ... encode features for subModel ...
  Vector v2 = new RandomAccessSparseVector(50000);
  double subScore = subModel.classifyScalar(v1);
  subModelEnc((byte[]) null, subScore, v2);
  ... encode other features ...
  ... train model ...
}
```

Here we assume that there's a first model already trained and available in the file sub-model.model. Then when training the second model, we build a feature vector to be used as input for the first model, and run that first model to compute a score. That score is then used as a feature for the second model.

15.4.2 *Tuning the classifier*

In addition to changing the data that's presented to the classifier learning algorithm, it's possible to change the algorithm or to modify the parameters that define the algorithm. The benefit of trying these approaches is that it provides you with a variety of ways to improve performance through guided trial and error.

TRY ALTERNATIVE ALGORITHMS

It's important to test different classifier algorithms to see which algorithm does best. Mahout supports naive Bayes, complementary naive Bayes, and SGD. There are also experimental implementations of random forests and support vector machine (SVM) available.

In addition to changing the entire class of model, it's possible to change the learning algorithm while keeping the same kind of model. In particular, SGD-based classifiers can operate the low-level learning algorithm with cross validation using `CrossFoldLearner`, or the self-adapting `AdaptiveLogisticRegression`, which adjusts the many learning parameters automatically. There's also the choice of just what the learning algorithm is optimizing, choosing from log likelihood, AUC, or mixtures of these.

Compare different algorithms based on performance, training time, and classification speed. For example, try both versions of the Bayes algorithms as well as `AdaptiveLogisticRegression`. If more than one model does reasonably well on accuracy metrics, decide between them based on how well they integrate with the rest of the system.

TUNE THE LEARNING ALGORITHM

Particularly with the low-level SGD learning algorithms, there are many knobs that can be turned to get different results. Most of these parameters are adjusted when constructing an `AdaptiveLogisticRegression`, `CrossFoldLearner`, or `OnlineLogisticRegression` using similar configuration methods. These methods are summarized in table 15.5.

Table 15.5 Configuration methods for SGD learning classes

Algorithm parameter	How configured	Online logistic regression	Cross fold learner	Adaptive logistic regression	Description
`prior`	Constructor	Yes	Yes	Yes	Sets the regularization function to encourage sparsity and avoid overfitting
`numFeatures`	Constructor	Yes	Yes	Yes	Specifies the size of the feature vector
`numCategories`	Constructor	Yes	Yes	Yes	Specifies how many target categories there are
`alpha`	Setter	Yes	Yes		Sets learning rate schedule
`decayExponent`	Setter	Yes	Yes		Sets learning rate schedule
`lambda`	Setter	Yes	Yes		Controls amount of regularization
`learningRate`	Setter	Yes	Yes		Sets learning rate schedule
`stepOffset`	Setter	Yes	Yes		Sets learning rate schedule

The prior, number of features, and number of target categories are configured in all SGD classes in the constructor. The `AdaptiveLogisticRegression` class controls the learning rate by running a number of `CrossFoldLearners` in parallel with different settings, and comparing the results obtained with each of them. This means that with `AdaptiveLogisticRegression`, the only other configuration settings needed are those used to control how often the learning rate is changed and how many different parallel `CrossFoldLearners` are used to test different settings. For the `CrossFoldLearner` and `OnlineLogisticRegression`, you need to specify the learning rate and also how the learning rate will decay over time. This is done using builder-style methods such as those shown here:

```
lr = new CrossFoldLearner(targetCategories, numFeatures, new L1())
          .lambda(currentLambda)
          .learningRate(initialLearningRate)
          .decayExponent(0.5)
          .stepOffset(2000);
```

Usually, decayExponent and stepOffset are used to control how the initial learning rate is decreased over time; setting alpha is much less common.

In addition to these settings, the CrossFoldLearner allows you to set the number of cross-validation folds it uses internally. The default value of 5 is acceptable in almost all cases. Using a lower value, such as 2 or 3, decreases the amount of data available for training more than most applications are willing to accept, and setting the value much higher than 5 incurs higher computation costs because it needs an independent learner for each fold.

The simplest approach to tuning all of these parameters is to just use AdaptiveLogisticRegression with the default settings. But this costs quite a bit in terms of computing power, because the AdaptiveLogisticRegression uses an internal evolutionary pool of 20 CrossFoldLearners, each of which internally maintains 5 OnlineLogisticRegression objects. This means you pay the cost of running 100 learning algorithms at the same time.

In practice, though, the penalty is much smaller than that because reading training examples from disk and encoding them as feature vectors often costs nearly as much as running the 100 learning algorithms. Moreover, in practical situations, a learning algorithm will be run with at least dozens and possibly hundreds or thousands of settings for these parameters. The AdaptiveLogisticRegression does all that automatically. As a side effect, data must only be encoded once. Bad options are cut off early, so no computational power is wasted on them.

15.5 Summary

You've now seen the major APIs in Mahout for computing classification quality metrics, and you've seen the basics of how to use these metrics. You know how common bugs affect these metrics and how to find and correct common problems with classification.

As the examples in this chapter have shown, problems with classifiers can be subtle. Unlike in most computer programming, even a serious error may not cause complete failure because learning algorithms often learn how to compensate for partial failures. This situation is both good and bad news, and it makes detailed quality testing important.

Evaluation isn't only an iterative process during the steps of building and tuning a high-performing classifier; it's also an important ongoing step after production launches. Evaluating performance over time is critical to maintaining performance. External conditions may change, making the model less useful, or operational errors in data collection can degrade a model's effectiveness. In any case, ongoing evaluation during production is key to being able to recognize the need to retune or retrain a model or opt for a new approach. These issues are explored in the discussion of deployment in the next chapter.

16 Deploying a classifier

This chapter covers

- Specifying the speed and size requirements of a classifier
- Building a large-scale classifier
- Delivering a high-speed classifier
- Building and deploying a working classification server

This chapter examines the issues you face in putting a classification system into real-world production. In previous chapters, we talked about the advantages of using Mahout classifiers when working with large data sets, how the classifiers are trained, how they work internally, and how you can evaluate them, but those discussions purposely omitted many of the ways that the real world intrudes when you need to put a classifier into production. In this chapter, we examine these real-world aspects as we explain how to deploy a high-speed classifier. We include tips on how to optimize feature extraction and vector encoding, describe how to balance load and speed requirements, and explain how to build a training pipeline for very large systems, such as those appropriate for Mahout classification. Finally, we provide an example of the deployment of a Thrift-based server that allows fully functional load-balanced classification.

If you're planning to use Mahout in production, there's a good chance that some or all of the following are true: you have a huge amount of training data, you have high throughput rates, you have fast response time requirements, or you need to integrate with a large pre-existing system. Each of these characteristics implies specific architectural and design considerations that you need to take into account as you build your system. Large data sets require efficient ingestion and organization; high throughput requires scalable and optimized code. Integration into large systems requires a simple and clean API that plays well as a networked service. Mahout offers a powerful way to deal with big data systems, but there are also ways to simplify or restate problems to make tools like Mahout even more powerful.

In the next section, we present an overview of how to design, build, and deploy a real-world large-scale classifier.

16.1 Process for deployment in huge systems

There are a few standard steps that will help ensure success when you deploy a Mahout-based classifier system. The key is to anticipate problems that may arise and then choose which ones you'll put your primary efforts toward preventing. The detailed strategies for coping with each of these issues are explained in sections 16.2 through 16.4; in this section we introduce the process of deploying a huge system.

> **NOTE** To understand the potential value of Mahout for your classification project, look for aspects of your system that are especially large, or where you need it to be especially fast. Finding these pain points will tell you whether or not Mahout is the right solution.

The deployment process can be broken down into these steps:

- Scope out the problem
- Optimize feature extraction as needed
- Optimize vector extraction as needed
- Deploy the scalable classifier service

Each of these steps is described in this section and then put into practice in the examples that follow.

16.1.1 Scope out the problem

The first step is to determine how large a problem you actually have and how fast your system has to run in both training and production classification. The size of your training data dictates whether feature extraction will allow use of conventional tools, like a relational database, or whether you'll need to put some effort into building a highly scalable feature extraction system using tools like Apache Hadoop. The specifics of size and speed requirements will be explored in section 16.2.

The allowable time for training will determine whether you can use a sequentially trained classifier such as SGD at all. Here's why. With a parallel algorithm and limited

time for training, you can add more machines if your system isn't fast enough to process larger amounts of data. But if you use a sequential training algorithm, you've got a budget of one computer for a fixed time. To make this work with large amounts of data, you may have to put extra effort into making the parsing and encoding of data particularly efficient.

The throughput and latency requirements for the production classifier will determine whether you need a distributed classifier and whether you need to do extensive precomputation and currying to maximize classification speed. For utmost throughput at some cost in latency, you may also decide to do batch classification.

16.1.2 Optimize feature extraction as needed

As data sizes get to be more than tens of millions of examples, it becomes more and more important to use scalable techniques such as MapReduce for feature extraction.

At small to medium scale—up to a few million examples—any technique will work, and proven data management and extraction technologies like relational databases will serve well.

As scale increases, you're likely to have to commit to a parallel extraction architecture, such as Hadoop, or a commercially supported system like Aster Data, Vertica, or Greenplum. If you choose to use Hadoop, you'll also have to decide what general approach you'll use to implement your feature extraction code. Good options include Apache Hive (http://hive.apache.org/), Apache Pig (http://pig.apache.org), Cascading (http://www.cascading.org/), or even raw MapReduce programs written in Java.

A key point here is that any downsampling should be done as part of feature extraction rather than vector encoding so that it can be done in parallel. Issues in real-world feature extraction for big data systems are examined in detail in section 16.3.

16.1.3 Optimize vector encoding as needed

If you have less than about 100 million training examples per classifier after feature extraction and downsampling, consider a sequential model builder such as SGD. If you have more than about five million training examples, you'll also need to optimize your data parsing and feature vector encoding code fairly carefully. This requirement may involve parsing without string conversions, multithreaded input conversion, and caching strategies to maximize the speed with which data can be fed to the learning algorithm.

For more than about 10 million training examples, you may want to consider using a `CrossFoldLearner` directly with predefined learning-rate parameters. If the problem is relatively stable, you may be able to use an adaptive technique to learn good learning-rate schedules and then just stick with what works well. The advantage of doing this over using an `AdaptiveLogisticRegression` is that 20 times fewer numerical operations are required in learning. The disadvantages are that the speedup isn't nearly as large as it sounds, and your learning chain will require more maintenance.

16.1.4 Deploy a scalable classifier service

In most applications, it's convenient to separate classifiers into a distinct service that handles requests to classify data. These services are typically conventional services based on a networked RPC protocol such as Apache Thrift or REST-based HTTP. An example of how to build a Thrift-based classification server is shown in section 16.4. This is particularly appropriate where many classification operations are done on each input example, such as where many items are tested for expected value.

In a few cases, the network round trip is an unacceptable overhead and the classifier needs to be integrated as a directly called library. This happens when very small-grained requests must be done at extreme rates. Because it's possible to achieve throughputs of over 50,000 requests per second with modern network service architectures, such cases are relatively rare.

16.2 Determining scale and speed requirements

The first step in building a classifier is to characterize the scale factors, including the amount of training data and the throughput rates that your system is required to meet. Knowing the size of the problem in various dimensions will in turn lead to several design decisions that will characterize how your system will need to be built.

16.2.1 How big is big?

If you're considering using Mahout to build a classification system, ask yourself these questions about your system and use the answers to determine which parts of your system will require big-data techniques and which can be handled conventionally:

- How many training examples are there?
- What is the batch size for classification?
- What is the required response time for classification batches?
- How many classifications per second, in total, need to be done?
- What is the expected peak load for the system?

Estimating peak load

To estimate peak load, it's convenient to simply assume that a day has 20,000 seconds instead of the actual 86,400 seconds that make up a day. Using an abbreviated day of 20,000 seconds is a rule of thumb that allows you to convert from average rates to peak rates at plausible levels of transaction burstiness.

Estimating long-term average transaction rates is usually pretty easy, but you need to know the peak rate to be supported when you design your system. Using this rule of thumb allows you to make a so-called Fermi estimate for the required number of transactions per second that your system will need to handle. Enrico Fermi was a renowned physicist who was remarkably good at making estimates like this, and his name has become attached to such back-of-the-envelope computations.

These values need not be estimated exactly. Typically, numbers that are within an order of magnitude are good enough for initial system scaling and architecture. As you move into production, you'll be able to refine the estimates for final tuning.

Once you have an idea of the number of training examples you expect to use, you can consider whether or not Mahout is appropriate for your project, and if so, which algorithms are likely to be a good choice.

IMPLICATIONS OF THE NUMBER OF TRAINING EXAMPLES

Typically, Mahout becomes interesting once you exceed 100,000 training examples. There is, however, no lower bound on how much data can be used. When you have 10 million or more training examples, most other data-mining solutions are unable to complete training, whereas above 100 million or a billion training examples there are very few systems available that can build usable models. The number of training examples, along with model types, are the primary determinants of the training time.

With a scalable system such as Mahout, you can extrapolate fairly reliably from smaller tests to determine expected training time. With sequential learning algorithms such as SGD, learning time should scale linearly with input size. In fact, many models converge to acceptable error levels long before all available input has been processed, so training time grows sublinearly with input size.

With parallel learning algorithms such as naive Bayes, learning time scales roughly linearly with input size divided by the number of MapReduce nodes available for learning. Because naive Bayes learning is coded as a batch system where all input is processed through multiple phases, it can't be stopped early. This means that even though a partial data set might be sufficient to produce a useful model, all training examples must be processed before a usable model is available.

CLASSIFICATION BATCH SIZE

It's common for a number of classifications to be done to satisfy a single request. This requirement often occurs because multiple alternatives need to be evaluated. For instance, in an ad placement system, there are typically thousands of advertisements that each have models to be evaluated to determine which advertisement to show on a web page. The batch size in this case would be the number of advertisements that need to be evaluated. A few applications have very large batches of more than 100,000 models.

At the other extreme, a fraud-detection server might only have a single model whose value needs to be computed, because the question of whether a single transaction is fraudulent involves only one decision: fraud or not fraud. This situation is very different from an advertisement-selection server, where a click-prediction model must be computed for each of many possible advertisements.

MAXIMUM RESPONSE TIME AND REQUIRED THROUGHPUT

The maximum response time determines how quickly a single batch of classifications must be evaluated. This, in turn, defines a lower bound on the required classification rate. This rate is only a lower bound because if there are enough transactions per second, it may be necessary to support an even higher rate.

Typically, it's pretty easy to build a system that supports 1,000 classifications per second with small classification batch sizes because such a rate is small relative to what current computers can do. In fact, the Thrift-based server described later in this chapter can easily meet this requirement. Until batch sizes exceed 100 to 1,000 classifications per batch, you should be able to support roughly that transaction rate, because costs will be dominated by network time. At batch sizes of 10,000 or more, response times will start to climb substantially. At larger batch sizes and with subsecond response times required, you'll probably need to start scaling horizontally or restructuring how the classifications are done.

16.2.2 Balancing big versus fast

One of the advantages of using Mahout is that the trade-offs between large data and fast operation are much looser than in nonscalable systems. That said, there is almost always a trade-off to some extent between these characteristics of a system, so you'll need to consider how to balance these needs. To do this, keep in mind that one aspect of your system may have different requirements than another. It's common to use different classifiers for different purposes in the same system.

Once you have the overall characteristics of your system in mind, you can begin to outline those aspects of your deployment that will require special attention and which can be implemented more quickly. The following sections outline the most important considerations in each phase of the deployment. These considerations generally apply either during training or after deployment, during classification.

AT TRAINING TIME

Running the large-scale joins necessary to create classifiable data often dominates all other aspects of training. You can use standard Hadoop tools for that, but care needs to be taken to make the joins run efficiently.

Downsampling negative cases can help enormously by making your training data much, much smaller. In fraud or click-through optimization, the number of uninteresting training examples typically vastly outnumbers the number of interesting ones (the frauds or clicks). It's critical to accurate learning that you keep all of the interesting records, but many of the uninteresting records can often be randomly discarded as long as you keep enough to still dominate the interesting records. This discarding is called downsampling and is usually done as part of the joining process.

Feature encoding often dominates training time after you have your training examples. Downsampling the training data helps reduce the encoding time, but there are still a number of things you can do to make the encoding process itself more efficient.

Through it all, it's important to remember that building good classification models requires many cycles of iterative refinement. You need to make sure your training pipeline makes these iterations efficient. Section 16.3 covers the architectural and tactical issues involved in building a training pipeline.

AT CLASSIFICATION TIME

When you come to integrating your classifier into your system, your overall classification throughput needs will drive your selection of what you need to optimize in the classifier. Most Mahout classifiers are fast enough that this will only matter in extreme systems, but you need to check whether you're in the extreme category. If not, you're ready to roll with the standard methods, but if you're extreme, you'll need to budget time to stretch the normal performance limits.

If you need to do small numbers of classifications per transaction and have only modest latency and throughput requirements, it should suffice to use a conventional service design with a single thread per transaction. REST-based services may be acceptable, or you may need to use a somewhat higher-performance network service mechanism like Thrift, Avro, or Protocol Buffers. Section 16.5 has an example of how to build such a service.

If you require a moderate number (10s to 1000s) of classifications per evaluation and have more stringent latency and throughput requirements, you'll need to augment the basic classification service with a multithreaded evaluation of multiple models or inputs, but otherwise the basic design should still suffice.

For extreme batch sizes with very large numbers of model computations per service transaction, you'll need to dive into the heart of the model evaluation code in order to divide the model evaluation into portions that can be evaluated ahead of time and portions that must be evaluated at classification time. Details of such optimizations are highly specific to your particular application and are largely beyond the scope of this book. An example of such an implementation using Mahout is given in chapter 17.

16.3 Building a training pipeline for large systems

Classifications systems need training data, and scalable classification systems can eat truly enormous amounts of data. In order to feed these systems, you have to be able to process even larger volumes of data in order to pick out the bits that you can use to train your classifier.

In order to handle these huge volumes of data, there are a few things to keep in mind. The first and most important is to match your techniques to your task. If you have less than about a million training examples, for instance, almost any reasonable technique will work, including conventional relational databases or even scripts that process flat-file representations of your data. At 100 million training examples, relational databases can still do the necessary data preparation, but getting the data out becomes progressively more and more painful. At a billion examples and beyond, relational databases become impractical, and you'll probably have to resort to MapReduce-based systems like Apache Hadoop.

A billion training examples sounds extreme, but it's surprisingly common to reach that level. Shop It To Me, the company we profile in chapter 17, for example, sends several million emails each day, each containing hundreds of items. As a result, they

generate several hundred million training examples per day, a number that rapidly turns into billions of training examples. The larger ad networks all have well above a billion ad impressions per day, each of which is a potential training example. A music streaming service with a million unique visitors per month can produce nearly a billion training examples each month.

No matter what your volume, your model-training data pipeline will need to support the following capabilities:

- Data records will have to be acquired and stored coherently, typically in time-based partitions.
- Data records will have to be augmented by joining against reference data of some kind.
- Data records will need to be joined to target variable values.
- Downsampling of training data may be useful and should be supported.
- Other kinds of filtering and projection beyond simple downsampling will be required.
- Training records will need to be encoded as training features.
- Trained models will have to be retained and associated with metadata describing how to encode input records.

The various parts of the training data pipeline can usually scale horizontally, but when it comes to training a model, some of the most easily deployed kinds of Mahout models aren't suitable for horizontal scaling for the training process. This means that at very high data volumes, early cleaning and processing of data can be done in a highly scalable fashion, but significant efforts may be necessary to make data encoding very efficient for when the model is actually trained.

This section will cover the most important aspects of how to build these capabilities into your data pipeline using Mahout and related technologies. We focus next on techniques that you can use at large scales on the assumption that you'll already be familiar with the techniques that you can use at smaller data scales.

16.3.1 Acquiring and retaining large-scale data

Training a large-scale model means that you're going to have to deal with really big sets of training data. As always, issues that are trivial with small-scale data often balloon to magnificent and dire proportions when working with large-scale data. Most of these problems arise from the fact that large-scale data has a significant amount of what is metaphorically called inertia. This inertia makes it hard to move or transform large-scale data and requires that you plan out what you intend to do. In the absence of direct experience with a new class of data, this planning is impossible to do effectively. Instead, you have to fall back to following standard best practices until you have enough experience with your data.

ACQUIRING DATA

The first problem with large data sets is acquiring them. This sounds easy, but as with all things to do with large-scale data resources, it's harder than it looks. The primary issue at this level is to keep the data relatively compact and to maintain natural parallelism inherent in the production of the data.

For instance, if you have a farm of servers each handling thousands of transactions per second, it's probably a bad idea to use a single centralized log server. Excessive concentration is bad because it leaves you liable to single-point failures, and because it's very easy for backpressure from the data acquisition system to adversely affect the operation of the system being observed. To avoid this problem, it's crucial to build load shedding into the data acquisition phase; it's preferable to lose data than for your production system to crash.

The second trick to acquiring large data sets is to make sure that you keep data intended for machines to read in machine-readable form. It's nice to have human-readable logs, but extracting data from such logs is a common source of errors because of the fairly loose format. It's better to keep some data in human-readable form for humans to read and to keep the data you want to process in machine-readable formats.

For large-scale data ingestion, there are a number of different data formats that can be used, but only a few that provide the necessary combination of compactness, fast parsing libraries, and, most important, schema flexibility. Schema flexibility allows you to evolve the data that you store without losing the ability to read old data.

The two most prominent encoding formats that combine these features are Avro from Apache and Protocol Buffers from Google. Both are well supported and both provide comparable features and flexibility. Storing data in comma- or tab-separated fields is common, but this approach quickly leads to problems as schemas evolve, due to confusion about which column of data contains which field. The confusion becomes especially severe when files with differing schemas must be handled concurrently.

PARTITIONING AND STORING DATA

As you acquire and store data, there are several organizing principles that need to be maintained to ensure that you wind up with a scalable system:

- Directories should generally contain no more than a few thousand files. Larger directories can cause scaling problems because opening and creating files will begin to take longer with very large directories.
- Programs should process as few files as possible, which can be achieved by not processing irrelevant data and by keeping files as large as practical. Avoiding irrelevant data limits the amount of data to be transferred, and using large files minimizes the time lost due to disk seeks, thus maximizing I/O speed.
- The directory structure used to organize files should support your most common scheme for selecting relevant training data. Organizing directories well allows you to find relevant files without sifting through all of your files.

- Files should not generally be more than about 1 GB in size to facilitate handling them. Files should be small enough that you can transfer them between systems in a reasonably short period of time. This limits the impact of network disruption by allowing you to repeat transfers quickly. About 1 GB is a happy middle ground.

Following these principles generally leads to storing data with top-level directories that define the general category of data and with subdirectories that contain data from progressively finer time divisions.

> **TIP** It's generally good practice to concatenate small older files into larger files that represent large enough time periods to avoid excessively many input files for subsequent processing. If fine-grained time resolution is required for recent time periods, then files might be kept for very short periods, such as five minute intervals, but beyond a few days, these should be accumulated into hourly and then into daily files. If the data for a single day totals more than 1–2 GB, keeping several files for each day is probably a good idea.

WORKING INCREMENTALLY

For the most part, feature extraction isn't something that requires aggregation over long periods of time. That means that once you have done the necessary joins and data reduction to generate the corresponding training data for an hour's worth of data, that training data will probably not change when the next hour of data comes in. This characteristic makes it eminently feasible to collect training data incrementally rather than doing all feature extraction for the entire period of interest each time training is done.

Likewise, this incremental processing makes it possible to accumulate extracted features for short periods of time and then concatenate them for longer-term storage. It's common, for instance, to accumulate hourly training data and concatenate these hourly training files into dailies that are then kept pretty much forever. This corresponds nicely to the way that the raw data is usually kept.

When your classification system is relatively new, the addition of new features and changes in downsampling will likely mean that you'll need to rebuild your training data files fairly often. Over time, the frequency of change typically decreases. Once this happens, creating training data incrementally is a big win because the lag between the arrival of data and the beginning of model training can be decreased.

16.3.2 Denormalizing and downsampling

It's common for the data that you acquire directly to require denormalization before it can be used to train a model, because the original data may be scattered across many different storage formats including log files, database tables, and other sources. In denormalization, data that could be less redundantly stored in separate tables connected by keys is joined together so that each record is self-contained without any references to external data, such as user profile tables or product description tables.

Because model training algorithms have no provision for resolving external references, their inputs have to be fully denormalized to be useful.

Denormalization is almost always done using some kind of join. With small to medium data sets, almost any method will suffice to do these joins, and tools such as relational databases are very useful. With large datasets, however, these joins become much more difficult to do efficiently. Moreover, the characteristics of your data must be taken into consideration to do these joins.

JOIN TARGET FIRST

Joins are also almost universally required to attach target variable values to predictor variables. The target values should be attached, if possible, before denormalization, because it's common to downsample training records depending on the value of the target variable. By downsampling before any subsequent joins, it's often possible to achieve substantial savings during training, because less data is processed with almost identical results.

IN-MEMORY JOINS

In some cases, training data needs to be joined to a secondary table with a limited number of rows. When this is the case, the secondary table can sometimes be loaded into memory, and the join can be done by consulting an in-memory representation of the second table.

The in-memory table is loaded in the `configure` method of the mapper or reducer. In-memory joins can be done inside a MapReduce program as part of either the mapper or the reducer, although when you're generating classification training data, map-side joins are more common.

In-memory joins are called *memory-backed joins* in some references, such as Jimmy Lin and Chris Dyer's book, *Data-Intensive Text Processing with MapReduce.*

MERGE JOINS

Where both data sets to be joined are large, a merge join may be possible. In a merge-join in a sequential, non-parallel program, two data sets ordered on the same key can be joined in a single pass over both data sets.

In a MapReduce program, the situation is a bit more complex because the two data sets are very unlikely to be split in such a way that each map can merge data correctly. It's possible, however, to build an index on one file while sorting it. The path of the indexed data is provided as side data so that as each map starts reading a split of the unindexed data, it can seek to near the right place in the indexed data and start merging from that point.

Merge joins are advantageous when data is naturally sorted, but it's often overlooked that if only one of the inputs is sorted and indexed, then a merge join can be done in the reducer instead of in the mapper. This choice allows you to use the MapReduce framework to sort the unindexed input. Whether this approach saves time relative to a full reduce join is something that can only be determined by experimentation.

FULL REDUCE JOINS

In terms of programming effort, the simplest way to join data is to do a full reduce join. This means that you simply read both inputs in the mapper and reduce on the key of interest. Such joins can be more expensive than merge joins because more data needs to be sorted in the Hadoop shuffle step.

When you do a full reduce join, it helps to have Hadoop sort the results so that the training records come after the data being added to the training records to denormalize them. Having Hadoop do the sort allows the reducer to be written in a fully streaming fashion. Because the shuffle and sort in Hadoop are highly optimized, a brute-force full reduce join may be as efficient as a merge join, if not more so, especially if you're doing a reduce side merge join.

If your training data is larger than the data it's being joined to, a full reduce join and any kind of merge join are likely to be comparable in speed.

16.3.3 Training pitfalls

Even assuming that your data is what you think it is, your joins are working correctly, and you're parsing your data correctly, there are a few pitfalls that are very easy to run into. The most subtle and difficult to diagnose is a target leak. Also common is the use of an inappropriate encoding, which leads to a semantic mismatch between what you intended and how the model sees the data. Let's look at these issues in more detail.

TARGET LEAKS

Keeping daily training files substantially facilitates time-shifted training, which is often necessary to avoid target leaks.

For instance, suppose one feature that you want to use in a targeting system is a clustering of users based on click history. If you use that clustering to train a model on the same time periods that you used to derive the clustering, you'll probably find that the resulting model will do exceedingly well at predicting clicks. What's happening, however, is that your click-history clustering is probably putting everybody with no clicks into different clusters than everybody with clicks. This division means that the user cluster is a target leak in the time period when the clustering was created.

A picture of how such a target leak can creep into your design is in figure 16.1. To fix this leak, you can't just delete the problematic variable. The user-history clustering could still be a valuable predictive variable. The problem here is simply that the model isn't being trained on new data.

Figure 16.2 shows how this can be improved by clustering data from early time periods and then training on more recent data unknown to the clustering algorithm. Moreover, the withheld test data can be from even more recent time periods.

Keeping data segregated by time makes holding back test data very easy. In this scenario, you would use data from day 5 as withheld data to evaluate model performance.

SEMANTIC MISMATCH DURING ENCODING

A rose by any other name would still smell as sweet, but with training data, an integer isn't always what it seems. In some cases, an integer represents a quantity such as the

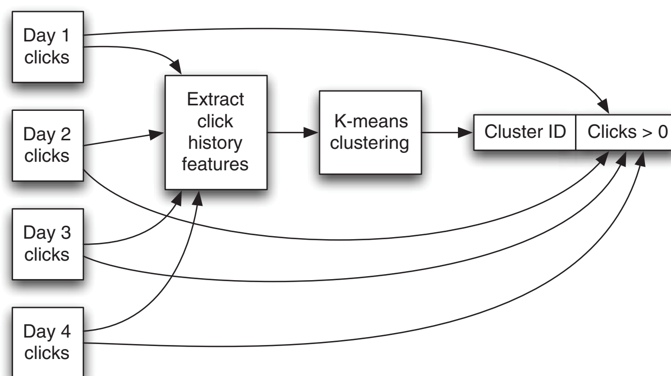


Figure 16.1 Don't do this: click-history clustering can introduce a target leak in the training data because the target variable (Clicks > 0) is based on the same data as the cluster ID.

number of hours or days between two events. In other cases, integers are used as identifiers for categories. The distinction between these cases has been covered in previous chapters, but in spite of the best intentions, it's still common for integer data to be treated incorrectly when training models. This mistake can be hard to debug, because a model built with incorrectly encoded data may actually be able to perform correctly a surprising amount of the time.

The key to diagnosing such problems is to review the internal structure of the resulting model to look for a few weights applied to each field. If you see lots of weights for particular values of a field, the field is being treated as an identifier for a categorical or word-like value. If you see a single weight for the field, it's being treated as a continuous variable. Whichever way the model is thinking of the field should match the way you think of the field.

The one exception to this sort of rule is when you have integer values that really are integers, but which only have a few distinct values. Treating such variables as having a categorical value may be useful even if they're more correctly treated as having a continuous value. Only experimentation can really determine if this will work for you, but whenever you have integers with a small number of values, this trick may be worth trying.

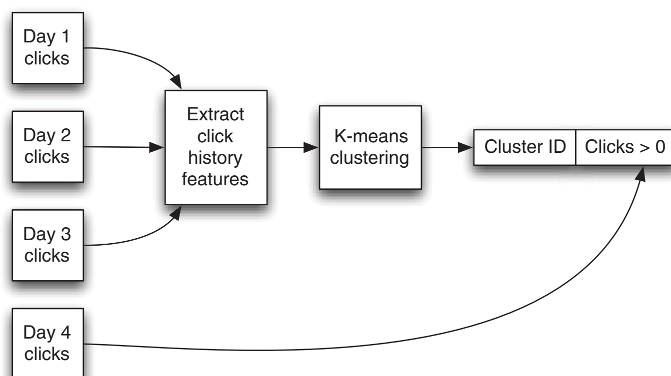


Figure 16.2 A good way to avoid a target leak: compute click history clusters based on days 1, 2, and 3, and derive the target variable (Clicks > 0) from day 4.

16.3.4 Reading and encoding data at speed

For many applications, the methods for reading, parsing, and encoding data as feature vectors that were discussed in previous chapters are good enough. It's important to note, however, that the SGD classifiers in Mahout only support training on a single machine rather than in parallel. This limitation can lead to slow training times if you have vats of training data. Look at a profiler and you'll see that most of the time used in the Mahout SGD model training API is spent in preparing the data for the training algorithm, not in doing the actual learning itself.

To speed this up, you have to go to a lower level than is typical in Java programs. The `String` data type, for instance, is handy in that it comes with good innate support for Unicode and lots of libraries for parsing, regex matching, and so on. Unfortunately, this happy generality also has costs in terms of complexity. It's common for training data to be subject to much more stringent assumptions about character set and content than is typical for general text.

Another major speed bump that training programs encounter is a copying style of processing. In this style, data is often processed by creating immutable strings that represent lines of input. These lines are segmented into lists of new immutable strings representing input fields. Then, if these fields represent text, they're further segmented into new immutable strings during tokenization. The tokens in the text may be stemmed, which involves creating yet another generation of immutable strings.

Doing all of the allocations involved in such a copy-heavy programming style costs quite a bit, and lots of people focus on reducing allocation costs by reusing data structures extensively. The real cost, however, isn't so much the cost of allocation, but the cost of copying the data over and over. An additional cost is due to the fact that constructing a new `String` object in Java involves creating the hash code for each new string. Computing the hash code costs almost as much as copying the data.

All of this copying provides an opportunity to increase speed enormously by working at a lower level of abstraction and avoiding all of these copies. In this lower-level style, if you do allocate new structures, you should try to make sure that they reference the original data rather than a copy. These small structures are also highly ephemeral, so the cost of allocating and collecting them is nearly zero.

As an example of a string copying style, here's a simple class that uses strings to parse tab- or comma-separated data. This class illustrates a simple, clear, and somewhat slow approach to parsing this kind of data:

```
class Line {
  private static final Splitter onTabs = Splitter.on(SEPARATOR);
  private List<String> data;
  private Line(String line) {
    data = Lists.newArrayList(onTabs.split(line));
  }
  public double getDouble(int field) {
    return Double.parseDouble(data.get(field));
  }

}
```

Obviously, important details are left out here, but the key idea is clear. The class splits a string representation of a line into a list of strings. This splitting and allocating of new strings requires that you copy the data around repeatedly. Moreover, by copying strings, you're copying two bytes per character.

The following listing shows how this code could be used to parse and encode data from a file containing CSV data.

Listing 16.1 Code to parse and encode CSV data

```
public static void main(String[] args) throws IOException {
  FeatureVectorEncoder[] encoder = new FeatureVectorEncoder[FIELDS];
 for (int i = 0; i < FIELDS; i++) {
    encoder[i] = new ConstantValueEncoder("v" + i);        Build one encoder
  }                                                          per field
  long t0 = System.currentTimeMillis();
  Vector v = new DenseVector(1000);
  BufferedReader in = new BufferedReader(new FileReader(args[1]));
  String line = in.readLine();
  while (line != null) {                                   Reset contents
    v.assign(0);                                            of vector
    Line x = new Line(line);                     Parse string
    for (int i = 0; i < FIELDS; i++) {
      encoder[i].addToVector((byte[]) null,                Encode each field
         x.getDouble(i), v);
    }
    line = in.readLine();
  }
  System.out.printf("\nElapsed time = %.3f s\n",
        (System.currentTimeMillis() - t0) / 1000.0);
}
```

This code reads each line, one at a time, and parses each line using the `Line` class shown earlier. For each field that's found, a feature encoder is used to add the field to the feature vector. Separate encoders are used for each field. When used to parse a million lines, each with 100 data elements, the string style of parsing and encoding shown in listing 16.1 takes 75 to 80 seconds running on a 2.8 GHz Core Duo processor. If you make a few changes, you can speed this up by a factor of at least 5. With the improvements we're about to describe, a program that does the same work takes well under 15 seconds.

SEGMENT BYTES, NOT CHARACTERS

The first step to speed things up is to use large-scale byte I/O, and not to convert bytes to `Strings`, and definitely to avoid copying pieces of the input over and over. In the `Line` class, each line is copied from the input, then copied again when it's converted to a string, and then copied again as the pieces are parsed out. All of this copying makes the program easier to understand and to debug, but it definitely slows things down quite a bit.

For many data files, avoiding the conversion to Java strings is a fine thing to do because the files generally have a limited data format that can be expressed in an

ASCII character set, where each character is encoded by a single byte code. Java characters are twice as large as bytes, and moving them around requires twice as much memory bandwidth. Even better, with byte arrays, you have the option to avoid copies almost entirely. If the fields in question contain numbers, using the normal Java primitives means that the fields will be parsed by routines that are capable of converting general Unicode strings into numbers. This conversion is harder than it sounds because basic digits appear many times in Unicode. If your data is all ASCII, the conversion can be done a lot more simply than by converting your data to Unicode and then converting such a general representation to a number.

Listing 16.2 shows an adaptation of the `Line` class called `FastLine` that uses a `ByteBuffer` to avoid copies. This code can also be found as an inner class in the `SimpleCsvExamples` program in the Mahout examples module. `FastLine` also parses numbers in a way that's highly specialized for parsing one- and two-digit integers encoded in an ISO-Latin character set.

Listing 16.2 Code for byte-level CSV parsing

```
private static class FastLine {
  private ByteBuffer base;
  private IntArrayList start = new IntArrayList();
  private IntArrayList length = new IntArrayList();
  private FastLine(ByteBuffer base) {
    this.base = base;
 }
 public static FastLine read(ByteBuffer buf) {
    if (buf.remaining() == 0) return null;
    FastLine r = new FastLine(buf);
    r.start.add(buf.position());
    int offset = buf.position();
    while (offset < buf.limit()) {
      int ch = buf.get();
      switch (ch) {
       case '\n':
         r.length.add(offset - r.start.get(r.length.size()) - 1);
         return r;
        case SEPARATOR_CHAR:
          r.length.add(offset - r.start.get(r.length.size()) - 1);
          r.start.add(offset);
          break;
        default:
       }
    }
    throw new IllegalArgumentException(
           "Not enough bytes in buffer");
   }

    public double getDouble(int field) {
     int offset = start.get(field);
      int size = length.get(field);
      switch (size) {
       case 1:
```

Annotations: Special collections avoid boxing; Remember a reference; Remember end of line is end of field; Note start of next field; Assume buffer has a full line

```
          return base.get(offset) - '0';
        case 2:
          return (base.get(offset)-'0') * 10+base.get(offset + 1) - '0';
        default:
          double r = 0;
          for (int i = 0; i < size; i++) {
            r = 10 * r + base.get(offset + i) - '0';
          }
         return r;
      }
  }
}
```

The `FastLine` class in listing 16.2 uses `IntArrayLists` from the Mahout collections package to store offsets and lengths. This has two effects. One is to avoid the boxing and unboxing of integers. A second effect is that fields are kept as references into the original byte array to avoid a copy.

The parsing of the file data is based on several strong assumptions that depend on the input data being somewhat constrained. First, there's an assumption that the `ByteBuffer` always has enough data in it to complete the current line. Second, the data is assumed to use Unix-style line separators consisting of a newline character with no carriage return character. Thirdly, it's assumed that only the ASCII subset of characters is used.

These assumptions allow considerable liberties to be taken, with the result that the time required for reading and parsing the data is less than one-third the time required for the same operations in the `String`-based code. `FastLine` is good, but it isn't the whole story.

DIRECT VALUE INTERFACE TO VALUE ENCODERS

Values can be encoded as vectors in a number of ways. For continuous variables, you can use the `ContinuousValueEncoder` and pass in the value as a string, as you saw in previous chapters. On the other hand, if you already have the desired value in the form of a floating-point number, you can pass in a null as the string and pass in the value in the form of the weight.

In the case of a direct byte parser, you can have access to the value of the field very quickly without converting to a string and then to a floating-point representation. This makes the shortcut use of the weight field on the `ConstantValueEncoder` attractive. The following code shows how this can be done.

Listing 16.3 Direct value encoding

```
public static void main(String[] args) throws IOException {
  FeatureVectorEncoder[] encoder = new FeatureVectorEncoder[FIELDS];
  for (int i = 0; i < FIELDS; i++) {
    encoder[i] = new ConstantValueEncoder("v" + i);
  }
  long t0 = System.currentTimeMillis();
  Vector v = new DenseVector(1000);
```

```
    ByteBuffer buf=ByteBuffer.wrap(
        FileUtils.readFileToByteArray(new File(args[1])));
    FastLine line = FastLine.read(buf);
    while (line != null) {
      v.assign(0);
      for (int i = 0; i < FIELDS; i++) {
        encoder[i].addToVector((byte[]) null,              <- Uses null string value
            line.getDouble(i), v);
      }
      line = FastLine.read(buf);
    }
    System.out.printf("\nElapsed time = %.3f s\n",
        (System.currentTimeMillis() - t0) / 1000.0);
  }
```

This code is similar to the string-oriented encoding code, except for the use of `FastLine` instead of `Line`. The same encoders are constructed and lines are still read, but from a `FastLine` instead of from a string level input. The encoding of each field is a bit different because the `FastLine` class converts the byte representation to a number, to avoid letting the encoder do the job.

The final version of the code using `FastLine` is about five times faster, as was mentioned earlier, but the effect of this direct value-encoding trick in isolation is even more impressive. For the million lines, ignoring the time required for data parsing, it takes about 4–5 seconds to encode all of the values when they're extracted from the byte representation, but it takes over 40 seconds to encode the values from string form.

The overall effect of these speedups is that raw data to be classified can be read, parsed, and encoded at a rate of about 15 MB/s without much effort. By using several threads for reading, the aggregate conversion rate can easily match disk transfer speeds for several spindles. With more elaborate encodings including lots of interaction variables, the cost of encoding will be higher, but these improvements will still have large impacts.

Such optimizations are worthwhile if you're running large modeling runs, especially if you're using a lower level `CrossFoldLearner` or `OnlineLogisticRegression` object directly, without the mediation of an `AdaptiveLogisticRegression`, because these lower-level learners are so fast that reading the training data can be the limiting factor on performance. On the other hand, using the more abstracted string-based methods allows you to write code that's much easier to write and debug and much less prone to surprises if your data files have oddities in them.

Once you have your data read (quickly) and converted (quickly), the next performance bottleneck is likely to be integrating your classifier into a server.

16.4 Integrating a Mahout classifier

Integrating a Mahout classifier is usually just a matter of creating a simple networked service. As such, the general issues of throughput, latency, and service updates that apply to all networked services apply to a Mahout classifier service. These issues can

take on a slightly different hue with Mahout classifiers than is customary because Mahout classifiers are often integrated into applications that require extreme speed. But to some extent, services based on Mahout classifiers are simpler than most services because they're stateless. This statelessness allows trivial horizontal scaling.

This section will cover how to plan and execute the job of integrating a Mahout classifier into a service-based architecture. Then in section 16.5 these ideas are pulled together into a complete working example classification server.

16.4.1 Plan ahead: key issues for integration

Although Mahout-based services have the advantage of being simple, you should still consider some key things when creating such a service in order to use Mahout to its full potential. These key issues include best practices about the division of responsibility for the client and server, checking production data for differences from training data, designing for speed, and handling model updates.

DIVIDING RESPONSIBILITIES

One of the critical architectural choices you'll face as you integrate classification models into a classification service is the division of responsibilities between the client and server sides of your system. A well-considered division can greatly enhance performance and will help future-proof your design.

In general, it's important to ensure that the server handles the encoding of features and evaluation of the model. There's a gray area, however, in how much feature extraction should be done on the client or server sides. For instance, if the client has a user ID and a page URL, but the server needs information from the user's profile and some content features based on the content of the page, then either the client or the server can do the necessary join against the user profile database and a page content or feature cache. Doing these join and feature-extraction operations on the client side will leave the model server with a much more circumscribed task and will likely result in a more reliable server. Doing these join and extraction operations on the server side will make certain kinds of caching easier and will hide the details of the model from the client a bit more.

Figure 16.3 illustrates how the responsibilities can be divided.

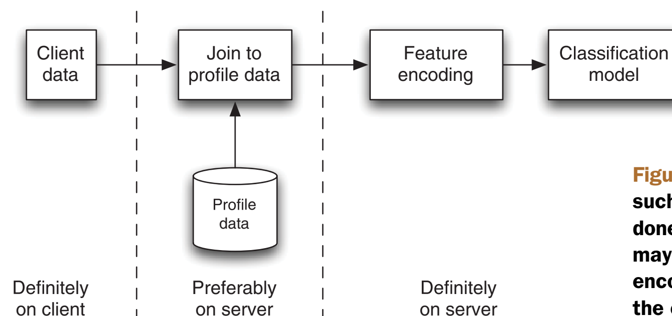


Figure 16.3 Joins with profile data, such as user profiles, are preferably done on the server, but these joins may be done by the client. Feature encoding should definitely be done on the classification server.

It's a good design principle for the model server to accept whatever form of data the client programs already have. If possible, the model server shouldn't require extensive data preparations to be done by the client unless they were already being done before the model service existed. A major exception to this principle is when providing the model server access to join resources would create a need for access to secured resources that are already available to the client. In such a case, joins that are necessary in preparation for feature encoding may have to be done by the client.

REAL-TIME FEATURES ARE DIFFERENT

One major issue with deploying a model into production is that the data that's given to a classifier in production is often quite different from the data that's reconstructed for training a classifier. These differences are generally not intentional, and they can lead to subtle bugs and apparent poor performance of the classifier system if not handled properly.

What sort of differences are you likely to encounter? Some of these differences are due to time-shifting processes, similar to target leaks. For instance, suppose you're trying to predict whether a user will purchase a particular product, and the predictor variables for the problem are based on the contents of the user's profile. In creating the training data for this model, it would be easy to collect user profiles as they are at the time the training data is collected rather than as they were at the time the purchase decision was made. If the completion of the purchase process results in additional information being stored in the user profile, any model built using the time-shifted profiles is likely to behave very poorly when given non-shifted profiles.

LOGGING REQUESTS

One of the best ways to detect when production data differs from training data is to log all classification requests even before the classification system is ready for deployment. After a short period of logging, you can use any technique you prefer to extract training data for the period of time in question, and compare the logged data with the extracted data. If they differ, the training data extraction process must somehow be changed to avoid extracting time-shifted training examples. Ultimately, it may be that only directly logged data can be used for training.

DESIGNING FOR SPEED

Speed considerations when building a classification system often go to paradoxical extremes. When a Mahout classifier is properly integrated, extreme speeds are definitely achievable. If your service is evaluating a single model against a single input record, and the classification server doesn't have to do any expensive joins, the model evaluation is likely to be so fast that network overheads will dominate. The speed problem will reduce to simply finding an access method that can drive high volumes of requests through the server.

For instance, a Thrift server running locally takes a bit less than 50–100 μs to do a server request, no matter how little the request actually does. Because a typical model evaluation requires no more than a thousand or so floating-point operations, it will finish in a few tens of microseconds at most, resulting in 90 percent of the time being

spent in overhead. The throughput will be a bit higher than this due to threading and network transit time, but it's clear that most of the time isn't spent on model evaluation. In such a system, the key place to put your effort is in optimizing network latency. To achieve the highest speed, you may need to integrate the model directly into your client code, avoiding server round trips entirely. This direct integration minimizes latency for each model evaluation, but it may substantially complicate model updates.

> **TIP** If you need to evaluate one example at a time, focus on decreasing network round trip times to minimize evaluation latency, and consider client-side model evaluation. For model updates, you may need some kind of server to simplify the distribution of model updates rather than depending on the file being accessible as a local file.

In the middle of the speed spectrum, some uses of classifier models require many models to be evaluated at a time, or allow many feature vectors to be sent together in a single server request. This batched evaluation approach can allow substantial increases in efficiency by virtue of the fact that network latency is amortized over more model computations. Latency isn't decreased relative to the case with a single evaluation per server request, but the amount of computation that can be done per request is increased dramatically.

> **NOTE** Doing lots of model evaluations per server request is a good thing, especially in terms of throughput. Many targeting systems require model evaluations for lots of items and thus allow this optimization.

At the extreme end of the speed spectrum, systems typically have enormous numbers of inputs to be evaluated. The computational marketing system described in the next chapter, for instance, requires that tens to hundreds of thousands of products be evaluated for each user, resulting in as many model evaluations as there are items. One request is made per user, and a feature vector is constructed using user parameters, item parameters, and user-item interactions. The models in that system are also fairly elaborate, involving hundreds or thousands of nonzero features for each item. In such cases, the evaluation time can be several hundred milliseconds, and computation time totally dominates all architectural decisions.

COORDINATING MODEL UPDATES

In a production system that's handling a stream of classification requests, it's fairly common for there to be an expectation of nearly 100 percent uptime. Because a classifier is normally stateless, this uptime requirement can be satisfied by placing a pool of classification servers behind a load balancer. Whenever the classification model is updated and released to production, it's necessary for the servers to load the new model and start using the new model for classification. In some cases, there is an additional requirement that model updates be made visible at essentially the same time across the entire pool of servers to maintain consistency among all servers.

Regardless of your exact requirements, we highly recommend using a coordination service to manage model updates and to provide a reliable list of live servers.

Among such services, Apache ZooKeeper is far and away the most popular, and we highly recommend it. ZooKeeper allows you to connect to a small cluster of servers that provide an API to access what looks a lot like an ordinary filesystem. In addition to very simple create, replace, and delete functions, however, ZooKeeper provides change notifications and a variety of consistency guarantees that simplify building reliable distributed systems even in the presence of server failures, maintenance periods, and network partitions.

A ZooKeeper-coordinated architecture for model updates can be quite simple, especially if exactly synchronized updates aren't necessary. In this architecture, ZooKeeper holds a model configuration. Classification servers request notifications about changes to this configuration, and while they have a live model, they maintain indicator files that tell classification clients which servers are handling requests. Classification clients interrogate ZooKeeper to determine which classification servers are available.

This data in ZooKeeper can be arranged in the following directory structure:

```
/model-farm/
   model-to-serve
   current-servers/
     10.1.5.30
     10.1.5.31
     10.1.5.32
```

In this directory structure, the /model-farm/model-to-serve file contains the URL of the serialized version of the model that all servers should be using. Each live server reads this file on startup and tells ZooKeeper to provide notifications of any changes. In addition, each server should poll this file every 30–60 seconds in case notifications couldn't be set, possibly because the file didn't exist when the server was started.

When ZooKeeper notifies the server that a change to model-to-serve has been made, the server should download the serialized form of the model and instantiate a new model. This new model should then be used to serve all subsequent requests. Once the new model is properly installed, the server should create a file in the /model-farm/current-servers directory with a name unique to the server, and it should contain, either in the name or file contents, information necessary for a client to send requests to the server. Such information might include a host name and port number. Some care is warranted here to deal correctly with the case of machines with multiple network interfaces.

Figure 16.4 contains a ladder diagram showing a typical time sequence for the deployment of a new model.

In the sequence depicted in figure 16.4, we assume that ZooKeeper starts with a model specification. When server1 starts, it loads the specification from ZooKeeper and then loads the model. When the model is fully loaded and ready to serve, server1 creates a file in ZooKeeper to indicate that it's ready. When server2 starts up a short time later, the same sequence ensues. When the control process updates ZooKeeper with a new model, notifications are passed to server1 and server2, and the model download and announcement process repeats.

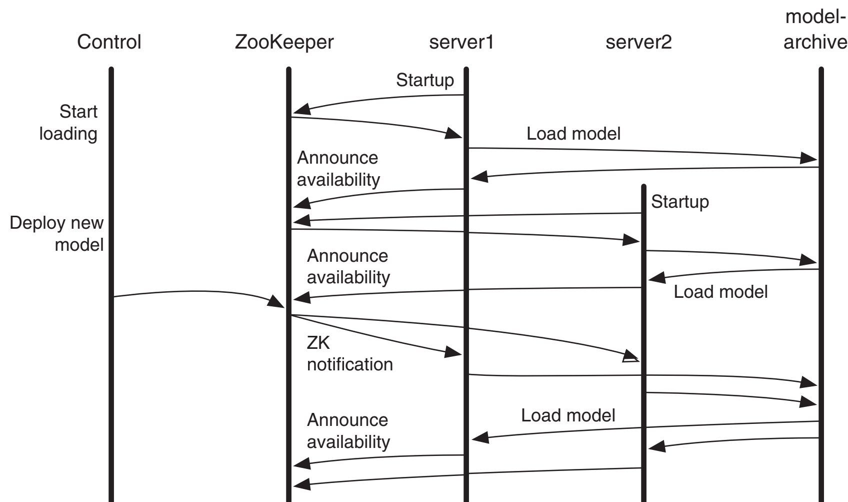


Figure 16.4 How model deployments are coordinated with ZooKeeper. The control process, server1, and server2 all communicate via ZooKeeper, which also notifies server1 and server2 when changes happen.

This process is simple to understand and easy to implement. It has significant limitations, however, in that all servers run the same model, and all servers update their models essentially simultaneously. This simultaneous access could conceivably result in degraded service across the entire cluster because servers will switch to using the new model whenever they have completed the load process, which in turn could cause a short time when different servers will give different answers. Similarly, a server may have somewhat slower response time immediately after a new model is loaded, so having all servers in such a degraded state simultaneously may not be desirable. Also, if the model file is large, having all servers read it simultaneously could cause problems due to network overload.

Here is a directory structure that shows how updates can be coordinated so that a more cautious model update can be done:

```
/model-farm/
   should-load/
      node-10.1.5.30
      node-10.1.5.31
      node-10.1.5.32
      node-10.1.5.33
   currently-loaded/
      node-10.1.5.30
      node-10.1.5.31
      node-10.1.5.32
      node-10.1.5.33
   model-to-serve/
      node-10.1.5.30
      node-10.1.5.31
      node-10.1.5.32
      node-10.1.5.33
```

In this directory structure, the top-level model-farm directory is the name of the overall model cluster. Below this, the should-load directory contains a single file for each server named with the address of the server. The content of each of these files is a list of URLs and MD5 hashes that can be used to retrieve and verify the content of the model. Parallel to the should-load directory is the currently-loaded directory; it contains a single ephemeral file created by each active model server that contains a list of the MD5 hashes for all models that are currently loaded on that server. Finally, another sibling directory called model-to-serve contains one file per server node that contains the hash of the single model that should be used for all incoming requests.

In operation, each model server maintains a watch on the should-load directory and the model-to-serve file. Changes in the should-load directory signal that a model should be loaded or dropped, whereas changes to the model-to-serve file signal that all subsequent requests be routed to the indicated model. Whenever a model is loaded or unloaded, the corresponding ephemeral file in currently-loaded is updated.

This directory structure allows considerably more flexibility than the previous model at the cost of implementation complexity. This flexibility allows new models to be loaded and staged on multiple machines before making them available essentially simultaneously. It also allows you to load and run a new model on only one server in order to test the stability of the new model before rolling it out to all of the servers. The use of MD5 hashes allows the reuse of URLs with new content and allows servers to verify that they have loaded the intended model.

In practice, you may need to extend this model to control multiple model server farms or you may want to automate how models are rolled out to progressively more machines. It's also common to have servers update their status files with indicators about how much traffic they can handle. Clients sending requests can use these indicators when picking a server to send a request to.

Also, keep in mind that even if you train your model using an `AdaptiveLogisticRegression`, which contains many models inside, you only need to save a single model from one of the underlying `CrossFoldLearners`. This is important because the `AdaptiveLogisticRegression` contains a large number of classifiers, each of which has a potentially very large coefficient matrix inside. With a large feature vector size, the size of a serialized `AdaptiveLogisticRegression` can actually be hundreds of megabytes.

16.4.2 Model serialization

As of version 0.4 of Mahout, the SGD and naive Bayes models are serialized in different ways. The SGD models share a helper class called `ModelSerializer` that handles the serialization and deserialization of all SGD models. In contrast, the naive Bayes models are only ever serialized as a side effect of the multiple files created during training. Deserialization of naive Bayes models isn't done using a single method, but rather by reading these multiple files back into memory explicitly.

In future versions of Mahout, it's likely that the ModelSerializer or a similar class will likely be extended to handle naive Bayes models as well as SGD-based models. Until then, the serialization and deserialization of naive Bayes models will remain a tricky and highly variable process. Until a more usable serialization interface is completed, this means that any deployment of a naive Bayes model should be done using the command-line interface we described in previous chapters rather than programmatically.

USING THE MODELSERIALIZER FOR SGD MODELS

The ModelSerializer class provides static methods that serialize models to files and strings. The use of the class is quite simple, as shown in the following snippet, where both a single model from an AdaptiveLogisticRegression as well as the complete ensemble model are saved to different files:

```
if (learningAlgorithm.getBest() != null) {
   ModelSerializer.writeBinary("best.model",
      learningAlgorithm.getBest().getPayload().getLearner());
 }
 ModelSerializer.writeBinary("complete.model", learningAlgorithm);
```

All of the different kinds of SGD models can be serialized using the same approach.

Reading a model from a file is just as easy, as shown here:

```
learningAlgorithm = ModelSerializer.readBinary(
              new FileInputStream("complete.model"),
              AdaptiveLogisticRegression.class);

OnlineLogisticRegression bestSubModel = ModelSerializer.readBinary(
        new FileInputStream("best.model"), OnlineLogisticRegression.class);
```

In this snippet, two uses of a ModelSerializer are shown. The first might be used to reload an entire AdaptiveLogisticRegression, complete with submodels, whereas the second can be used to load a single OnlineLogisticRegression, which might be one of the component models in an AdaptiveLogisticRegression. Note how you have to supply the class to the readBinary method so that it can know what kind of object to construct and return.

When serializing SGD models for deployment as classifiers, it's usually best to only serialize the best-performing submodel from an AdaptiveLogisticRegression. The resulting serialized file will be 100 times smaller than would result from serializing the entire ensemble of models contained in the AdaptiveLogisticRegression object. In addition, when data is to be classified, the individual model is more suited to deployment because only the best submodel from the AdaptiveLogisticRegression object would be used, and all of the other models would be ignored.

On the other hand, if you're serializing a model so that you can continue training at a later time, serializing the entire AdaptiveLogisticRegression is probably a better idea.

16.5 Example: a Thrift-based classification server

Putting all the bits of a working classification server together can be a bit daunting. To help with that, this section contains a full working example that illustrates all that we've talked about. We've written a simplified, but complete, classification server that implements the simpler model deployment scheme described in section 16.4. This server provides all of the basic capabilities needed for a fully functional load-balanced classification service, including some administrative capabilities.

Communication between the classification client and server will be handled in this example using Apache Thrift (http://thrift.apache.org/). Thrift is an Apache project that allows client-server applications to be built very simply.

The coordination of multiple servers and between server and client will be handled in this example using Apache ZooKeeper (http://zookeeper.apache.org/). In this example, ZooKeeper holds instructions for all of the classification servers about which model to load, as well as status information from all of the servers so that the client can tell which servers are up and which model they're serving. This server coordination structure is shown in figure 16.5.

This arrangement allows the classification server to know where to find the model definition and allows the client to find all active servers.

When a server comes up, it queries ZooKeeper to find out what model to load. After loading that model, it announces to the world that it's ready for traffic by writing a file into ZooKeeper. Whenever the file indicating which file to load is changed, ZooKeeper notifies all servers of the change.

When a client wishes to send a query to a server, it can look into ZooKeeper to find out which servers are currently providing service and pick one server at random.

The structure of the ZooKeeper files that coorTdinate everything is as follows:

```
/model-service/
   model-to-serve
   current-servers/
     hostname-1
     hostname-2
```

The file called model-to-serve contains the URL of the model that should be loaded and used for classification by all servers. These servers will maintain a watch on this file so that they're notified as soon as it changes. When they see a change, they'll load the file to get the model URL, and then they'll reload the model. As soon as any model is loaded by a server, it will create an ephemeral file in the current-servers directory with the

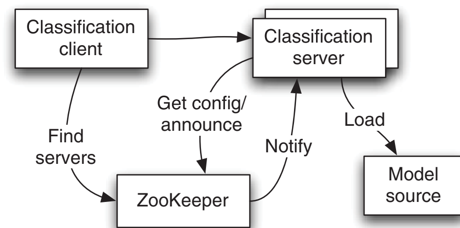


Figure 16.5 ZooKeeper serves as a coordination and notification server between the classification client and server.

name of the server. Because this file is ephemeral, it will vanish within a few seconds if the corresponding server ever crashes or exits.

The code for the main class of this model server is shown in listing 16.4. It mostly contains the setup code for the Thrift server layer, but note how a timer is used to schedule periodic checks in ZooKeeper. This should be completely redundant, but it's a good example of belt-and-suspenders coding style. ZooKeeper should notify you of any changes, but checking in now and again makes sure you'll catch up if something gets lost due to a programming error.

Listing 16.4 Main program for the classification server

```
public static final String ZK_BASE = "/model-service";
public static final String ZK_CURRENT_SERVERS =
               ZK_BASE + "/current-servers";
public static final String ZK_MODEL = ZK_BASE + "/model-to-serve";
private final TServer server;
private final Logger log =
               LoggerFactory.getLogger(this.getClass());
private final ZooKeeper zk;
private final Ops modelHandler;
private Timer timer;
private String currentUrl = null;
private int version;
private Watcher modelWatcher = new Watcher() {
                                                          ❶ Details in listing 16.5
  ...
}
public Server(int port)
      throws TTransportException, IOException,
                      InterruptedException, KeeperException {
  zk = new ZooKeeper("localhost", 2181, null);              Connect to local ZooKeeper
  modelHandler = new Ops();                                 Create classifier internal service
  timer = new Timer();                                      Retry model load every 30 seconds
  timer.scheduleAtFixedRate(new TimerTask() {
  @Override
  public void run() {
   modelWatcher.process(null);
  }
},  0,  30000);
socket = new TServerSocket(port);                            Set up thrift server
Classifier.Processor processor = new Classifier.Processor(modelHandler);
TProtocolFactory protocol = new TBinaryProtocol.Factory(true, true);
server = new TThreadPoolServer(
    new TThreadPoolServer.Args(socket).processor(processor));
log.warn("Starting server on port {}", port);

  server.serve();                                           Start server
}
public static void main(String[] args) throws IOException,
               TtransportException, InterruptedException, KeeperException {

  new Server(7908);
}
```

Most of the action is in the `modelWatcher` object ❶. This object is an implementation of a ZooKeeper `Watcher`, and it's invoked whenever a watch is triggered by the modification of the model-to-serve file. Here's the source code for the `modelWatcher`.

Listing 16.5 The watcher object loads the model and sets model status

```
private Watcher modelWatcher = new Watcher() {
  @Override
  public void process(WatchedEvent watchedEvent) {
    String hostname = null;
    try {
   hostname = InetAddress.getLocalHost().getHostName();
 } catch (UnknownHostException e) {
    }
 if (hostname == null) {
  log.error("Must have hostname ... exiting");
      System.exit(1);
 }
 String url = null;
 try {
   Stat stat = new Stat();
   byte[] urlAsBytes = zk.getData(ZK_MODEL, modelWatcher, stat);
   int latestVersion = stat.getVersion();
   url = new String(urlAsBytes, Charsets.UTF_8);
   if (currentUrl == null || latestVersion != version) {

     URL modelUrl = new URL(url);
     log.warn("Loading model from " + modelUrl);
     AbstractVectorClassifier model = ModelSerializer.readBinary(
         modelUrl.openStream(),
         OnlineLogisticRegression.class);
    try {
      zk.create(ZK_CURRENT_SERVERS + "/" + hostname,
                modelUrl.toString().getBytes(Charsets.UTF_8),
                ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL);
     } catch (KeeperException.NodeExistsException e) {

      zk.setData(ZK_CURRENT_SERVERS + "/" + hostname,
                modelUrl.toString().getBytes(Charsets.UTF_8), -1);
     } catch (KeeperException e) {
      log.error("Couldn't write server status file");
     }
     modelHandler.setModel(model);
     currentUrl = url;
     version = latestVersion;
     log.info("done loading version " + version);
  }
   return;
} catch (KeeperException.NoNodeException e) {
  log.error("Could not find model URL in ZK file: " + ZK_MODEL, e);
  return;
} catch (KeeperException e) {
  log.error("Failed to load model due to ZK exception", e);
} catch (InterruptedException e) {
  log.error("Operation interrupted should never happen", e);
```

❶ Get URL from ZK

Check for changed URL

❷ Tell ZK model server has loaded

❸ Update pre-existing file

Throw error if no data in ZK

```
} catch (IOException e) {
  log.error("Failed to load model from " + url, e);
}
log.warn("Clearing current URL due to error");
currentUrl = null;
version = -1;
  }
};
```

A lot of this code exists to handle exceptional conditions, but the basic outline is quite simple. The backbone of the code is in the calls to `zk.getData` ❶, `zk.create` ❷, and `zk.setData` ❸.

The way the code works is that it attempts to read the URL of the model from ZooKeeper. If a URL is read but there is no current model, or if the version of the ZooKeeper file containing the URL is different from the one for the current model, the model is reloaded and the status file for this server is updated with the model URL. The update is done by first trying to create the status file, and if that fails to update the status file.

If the original read of the URL failed due to the model file being missing, a message is logged and a relatively normal return ensues. Other errors cause the server to null out the URL of the current model to make sure that it reloads the model at the next opportunity.

The actual handling of classification requests is done by a Thrift server. The interface is defined in Thrift IDL as shown here:

```
namespace java com.tdunning.ch16.generated
service Classifier {
  list<double> classify(1: string text)
}
```

This interface definition declares that the server will accept classification requests. Each request will return a list of scores, one for each possible value of the target variable.

The interface defined by the preceding IDL is implemented in the `Ops` class, which is shown next.

Listing 16.6 Implementation of classification service

```
public class Ops implements Classifier.Iface {
  private static final int FEATURES = 10000;

  private static final FeatureVectorEncoder enc =
         new TextValueEncoder("text");
  volatile AbstractVectorClassifier model;

  public Ops() {}

  @Override
  public List<Double> classify(String text) throws TException {
    Vector features = new RandomAccessSparseVector(FEATURES);
    enc.addToVector(text, features);
```

```
    Vector r = model.classify(features);          <-- Classify text

    List<Double> rx = Lists.newArrayList();
    for (int i = 0; i < r.size(); i++) {          <-- Convert to
      rx.add(r.get(i));                               output format
    }
    return rx;
  }

  public void setModel(AbstractVectorClassifier model) {   <-- Provide a hook
    this.model = model;                                        for ZooKeeper
  }                                                          (1) watcher
}
```

The classify method classifies a feature vector using the currently loaded model. This method is called by the Thrift server when a client request is received and encodes the text passed to it in the style of the TrainNewsGroups program in chapter 15. The model is then used to compute a vector of scores, and these scores are returned.

The Ops class also defines a setModel (1) method that the ZooKeeper watcher class can use to load a new model when a model change is observed.

16.5.1 Running the classification server

To see this server in action, you can use the command-line interface to ZooKeeper to instigate behaviors. Before you start, though, you need a model to work with. TrainNewsGroups, from listing 15.3 in chapter 15, is a handy program to get such a model from because it writes a copy of a model to /tmp every few hundred training examples.

The process of setting up and running the TrainNewsGroups components is described in the README file with the source code.

When you run the TrainNewsGroups program, the result is a mass of output. The critical bits of the output are shown here:

```
[INFO] [exec:java {execution: default-cli}]
11314 training files
0.62 189992.00 ... 7000  -1.017 81.02 none
0.64 189992.00 ... 8000  -0.898 84.77 none
0.75 189997.00 ... 10000 -0.948 84.71 none
...
```

As a side effect, you get model files stored in /tmp:

```
$ ls -l /tmp/
-rw-r--r--  1 ...  1680247 Nov 21 01:16 news-group-1000.model
-rw-r--r--  1 ...  1680247 Nov 21 01:16 news-group-1200.model
-rw-r--r--  1 ...  1680247 Nov 21 01:16 news-group-1400.model
-rw-r--r--  1 ...  1680247 Nov 21 01:16 news-group-1500.model
...
-rw-r--r--  1 ...  1680247 Nov 21 01:19 news-group.model
$
```

The files with names like news-group-*.model contain the best model at each stage of learning.

Once you have some model files, you can start a local copy of ZooKeeper as described in the instructions in the README file from the Chapter 16 source code.

With ZooKeeper running, you can start the classification server:

```
...
10/11/21 01:36:06 INFO zookeeper.ClientCnxn: Socket connection established to
     localhost/fe80:0:0:0:0:0:0:1%1:2181, initiating session

10/11/21 01:36:06 WARN ch16.Server: Starting server on port 7908
...
10/11/21 01:36:07 ERROR ch16.Server: Could not find model URL in ZK file: /
     model-service/model-to-serve
10/11/21 01:36:08 WARN ch16.Server: Starting server on port 7908
...
```

This is best done in a separate window from the one used to start ZooKeeper so that you can separate the different log lines as they come out.

At this point, the classification server is running, but there is nothing in ZooKeeper to tell it what model to load and use to classify requests. To put the right things into ZooKeeper, you can use the ZooKeeper command-line interface, as shown next. Again, a new window is helpful to keep things separated.

```
$ ~/Apache/zookeeper/bin/zkCli.sh  <<EOF
create /model-service ""
create /model-service/model-to-serve \
           file://localhost/tmp/news-group-1500.model
quit
EOF
```

These commands should produce lots of output, most importantly including lines like this:

```
[zk: 1] Created /model-service
[zk: 3] Created /model-service/model-to-serve
```

As you were talking to ZooKeeper, the classification server you started earlier was checking every 30 seconds to see if ZooKeeper was correctly set up. Each time it did so, it would emit the same error message about not being able to find the model URL. On the cycle after you finally create the /model-service/model-to-serve file, however, it will produce a different message:

```
10/11/21 01:45:04 INFO ch16.Server: Loading model from file://localhost/tmp/
     news-group-1500.model
10/11/21 01:45:04 INFO ch16.Server: model loaded
```

At this point, the classification server is up and running. You can watch it respond to configuration commands in ZooKeeper by changing the content of the model-to-serve file:

```
~/Apache/zookeeper/bin/zkCli.sh  <<EOF
set /model-service/model-to-serve file://localhost/tmp/xxx.model \
quit
EOF
```

Almost instantly after doing this, you'll see the server output a warning that it can't find this bogus model. You can put things right again with this command:

```
~/Apache/zookeeper/bin/zkCli.sh  <<EOF
set /model-service/model-to-serve \
      file://localhost/tmp/news-group-1500.model
quit
EOF
```

Almost instantly, the classification server will respond with a confirmation that it has loaded the model. If you had 10 model servers running on different machines, they would all respond just as quickly.

16.5.2 Accessing the classifier service

On the client side, matters are even simpler. The following listing shows how the client can get information from ZooKeeper to find out what servers are live and then request one of them to do the desired classification.

Listing 16.7 Accessing the sample classification server

```
public class Client {
  public static void main(String[] args) throws TException, IOException,
        InterruptedException, KeeperException {
     ZooKeeper zk = new ZooKeeper("localhost", 2181, null);
     List<String> servers =
              zk.getChildren(Server.ZK_CURRENT_SERVERS, false, null);
     if (servers.size() == 0) {
       throw new IllegalStateException("No servers to query");
     }
     int n = new Random().nextInt(servers.size());
     String hostname = servers.get(n);
     Connection c = new Connection(hostname, 7908);
     List<Double> result1 = c.classify("this is some text to
    classify");
     System.out.printf("%s\n", result1);
     List<Double> result2 = c.classify("Given that the escrow " +
              "keys are generated 200 at a time on floppies, why\n" +
              "not keep them there rather than creating one huge" +
              " database that will have to\n" +
              "be guarded better than Fort Knox.");
     System.out.printf("%s\n", result2);
     c.close();
  }
}
```

1. Get list of live servers
2. Pick server at random
3. Send request and print result

The first thing that the client does is contact ZooKeeper to get a list of the live servers 1 and picks one at random 2. After that, it connects to the randomly selected server and requests the server to classify two bits of text 3.

The output of the client program should look like this when you run it, with some bits omitted here for clarity:

```
$ mvn exec:java -Dexec.mainClass="mia.classifier.ch16.Client"
...
[INFO] Preparing exec:java
...
10/11/21 20:15:26 INFO zookeeper.ClientCnxn: Opening socket connection to
     server localhost/0:0:0:0:0:0:0:1:2181
...
[0.03964640884739735, 0.07565438894170683,
0.05901603385987076, 0.05114326116258236,
0.050421016895119145, 0.044210065098318506,
0.04114422304836256, 0.05402402583820334,
0.05281506587810628, 0.04227194810117395,
0.07788774620593823, 0.048761793702438536,
0.03998325394587257, 0.03846382850793468,
0.04373069260414459, 0.04980658400349993,
0.04722614673439341, 0.04948828603244314,
0.04477719025270829]
[0.030532890514017766, 0.25984369795811524,
0.03596784998912587, 0.05321219256232682,
0.02395166487009193, 0.033935125988576176,
0.03729692334981463, 0.04035766767461885,
0.05067830734673832, 0.03391626207189733,
0.07418249664999789, 0.06666407060825316,
0.0424634477415305, 0.01074681122380413,
0.05212145428368258, 0.030600556819694907,
0.025032370110624636, 0.02760955132545441,
0.02696366801467259]
[INFO]
 ------------------------------------------------------------------------
[INFO]
 BUILD SUCCESSFUL
...
$
```

First request

Second request

The output from the two requests appears as a list of numbers, each of which is the score of a different newsgroup. The second request is the more interesting one because it has text that was extracted from an actual newsgroup posting. That request also shows a significantly higher score for the second newsgroup, as you would expect when giving a classifier some data it can make a decision about.

Clients written like this example inherently balance the load across multiple servers. Moreover, a slightly more advanced client that's written to perform multiple queries in sequence can maintain a watch on the current servers directory on ZooKeeper. Whenever the watch is triggered, the client can pick an alternative server. This will cause immediate load rebalancing as soon as any new server comes online or shortly after a live server exits or crashes.

This Apache Thrift-based server example isn't quite ready for prime time, but you should be able to take it as a skeleton for many kinds of real-world production deployments. A key aspect of the design of this system is the use of ZooKeeper for coordination and notification between the classification client and server in a system with

multiple servers. This approach allows the classification server to scale quickly and transparently and to change which models are live at any time.

16.6 Summary

Congratulations! You've successfully navigated the third and final stage in Mahout classification, deploying a large-scale classifier. Now you know how to build and train models for a classification system, how to evaluate and fine-tune your models to improve performance, and how to deploy the trained classifier in huge systems. Some key lessons learned from this chapter regarding deployment include the importance of fully scoping out the project, paying particular attention to any parts of your system that, as designed, could exceed what Mahout can easily do. Discovering these potential difficulties ahead of time gives you the chance to modify your design to avoid the problems or to budget time to stretch those boundaries.

Often the most intense aspect of deploying a large-scale system is building the training pipeline. In systems of the size you'll tackle with Mahout, the pipeline may offer some substantial engineering challenges. One key to doing this successfully is what you have learned about doing extremely large-scale joins to implement the pipeline in parallel.

The chapter also showed you some of the pitfalls to avoid: namely, how not to produce target leaks or create a semantic mismatch. We showed several subtle ways that these leaks can occur and how you can avoid them.

The final lessons you have learned involve the service design. In this chapter you have seen the core design of a robust classification service based on standard technologies like ZooKeeper and Thrift. If you keep the spirit of this service design, you should be able to deploy high-performance classification services reliably and repeatably. But don't be fooled by the simplicity of the design presented in this chapter. Much of the simplicity, robustness, and reliability of this system are provided by building on the firm foundation of ZooKeeper. Reflect carefully before you diverge too far from this basic design idea.

As you saw, deploying even a well-designed classifier is a complicated task when it's integrated into a larger system. We'll conclude our exploration of classification in the next chapter, which puts all of this together in presenting a real-world case study of a large-scale classification system used by an online marketing company, ShopItToMe.

17 Case study: Shop It To Me

This chapter covers

- Speed and scale considerations for the classifier system
- How the training pipeline was constructed
- Restructuring classification for very high throughput

So far, the chapters on classification have presented an overview of classification plus detailed explanations of what it takes to design and train a Mahout classifier, evaluate the trained models in order to adjust performance to the desired level, and deploy the classifier into large-scale systems. This chapter puts all those topics into practice in a case study drawn from a real-world online marketing company, called Shop It To Me (http://www.shopittome.com/), that selected Mahout as their approach to classification. You'll examine the issues their small engineering team faced in building and deploying a high-performance Mahout-based classifier and see the solutions they came up with.

Chapter 16 focused on the scale requirements of the very large systems that are best served by Mahout classification. Similarly, the case study presented in this

chapter deals with large data sets, but it also affords a look at a system with speed requirements that are extreme, even for Mahout systems. Because of the scale and especially the speed required, some of the solutions the team designed required substantial innovations. Overall, the system described in this chapter shows how you can push Mahout classifiers farther than appears possible at first glance.

After introducing the Shop It To Me system, we'll look at how the outbound email system works and describe the data that the system produces. The goal of the case study classifier is presented along with the steps taken in training the model. Then we examine how the Shop It To Me team made the training and model computations go fast enough to meet their business requirements.

Finally, we bring together the lessons to be learned from this case study so that you can apply what you learn here to projects of your own.

17.1 Why Shop It To Me chose Mahout

The Shop It To Me marketing case study will show why Mahout can be ideal to solve problems of scale and speed that don't yield well to other approaches. But in order for you to appreciate the challenges described in the case study, it's helpful to have some background information about the company. Let's look at what Shop It To Me does, what they wanted a classification system to do, and why they chose Mahout to build that system.

17.1.1 What Shop It To Me does

Shop It To Me is a free online personal shopper service that provides email notifications about sales (SaleMails) of interest to their customers. Their mission is to effectively connect a shopper with a list of sale items that are appropriate to the tastes of the customer and to do so in a timely manner. The latter is particularly important, because the business often deals with products that are temporarily discounted, and if options to click aren't delivered promptly to the customers, the sale items will disappear before the shopper can shop. In short, the business mission of the company is to show people what they want to see.

Shop It To Me has relationships with hundreds of retailers that allow the company to know when items will go on sale. With over 3 million customers to date, Shop It To Me makes over 2 billion product recommendations per month, in order to construct the SaleMails that they send out to customers.

17.1.2 Why Shop It To Me needed a classification system

You may be wondering, "Why do classification?" for a business mission that's essentially based on a form of recommendation; that is, predicting which sale items particular customers will want to see. The reason to choose classification is that the Shop It To Me situation is a case where normal recommendation methods don't work well. Recommendation systems work well with items that are seen by many users. Classic examples include movies and music. People can be seen as related to each other by noting that they interact similarly with the same items, and that similarity can be used

to predict other preferences. Most importantly, the items being recommended will still be there tomorrow.

This persistence isn't true for the products being recommended to customers by Shop It To Me. The best sale items are, literally, here today and gone tomorrow (if not later today). The sale items may go away faster than feedback on who is clicking on the item can even be obtained, much less used productively. This ephemeral nature of items means that the recommendation system has to take into account the characteristics of items in the training data, such as brand, price, or color that will be shared with other items that appear later. To do this requires a different approach: the use of classification to make predictions that operate as part of a system that acts like a recommendation system. At Shop It To Me, available data includes the personal history of the users and the characteristics of the items to be recommended. These form the predictors. There's also a good target variable as well: whether the user will click on an item.

In order to increase the utility of these recommendations, Shop It To Me has built an advanced classification system to help predict exactly which products will appeal to particular users. When constructing a SaleMail for a particular user, this classification system can compute the value of a classification model trained to predict whether this user will click on an item for each of hundreds of thousands of possible sale items. By ranking items according to the predicted probability of user interest, Shop It To Me can build SaleMails of particular interest to users.

17.1.3 Mahout outscales the rest

Building this system posed some particular challenges. Not only are there billions of training examples, but the required classification speed is fairly stupendous. A back-of-the-envelope calculation shows that millions of users need model evaluations for hundreds of thousands of items, giving hundreds of billions of evaluations. In order to meet the constraints of the email construction and delivery pipeline, these evaluations must be completed in just a few hours. This gives a raw classification rate requirement of tens of millions of classifications per second, which is, as they say in the old country, a lot.

To meet these severe requirements with reasonable-sized computing infrastructure, Shop It To Me has had to do some very interesting engineering. Several systems were used to produce prototype classifiers, including R (http://www.r-project.org/) and Vowpal Wabbit (https://github.com/JohnLangford/vowpal_wabbit/wiki), an open source SGD classifier sponsored by Yahoo. With R, simply handling the training data volumes and getting good integration with the existing Ruby/JRuby infrastructure at Shop It To Me proved very difficult. Vowpal Wabbit was able to handle the data volumes for training, but integrating a Vowpal Wabbit classifier well was also difficult.

Mahout, on the other hand, was able to handle the training volume and integrated well with the existing infrastructure. Mahout provides key portions of the Shop It To Me system, aspects of which are described in this chapter. Mahout isn't the preferred solution for every project, but it proved to be the best choice for the engineering challenges of Shop It To Me.

NOTE Some details of Shop It To Me's system design, variable extraction, and model performance are proprietary and have been omitted or glossed over.

The system outline here is broadly descriptive and doesn't contain an exact representation of all aspects of the Shop It To Me system. Enough details are included, however, to allow you to understand the scaling issues and solutions. Those issues start with the structure of the email system itself.

Using Mahout from Ruby

Shop It To Me is a Ruby shop, but they chose Java-based Mahout as a platform for modeling. This may seem like a contradiction, but as the Shop It To Me engineers determined, Java integrates remarkably easily into a system composed of Ruby-based web services and processes. One of the main reasons for this is that JRuby allows nearly transparent calling from Ruby code into Java without losing significant performance relative to standard Ruby. That allows classification services to be built using standard Ruby idioms and engineering practices.

On the other hand, model training at Shop It To Me is done using Cascading integrated into a Ruby workflow defined in Rake. The Java nature of Cascading is hidden in this system by a special-purpose domain-specific language (DSL) for writing Cascading flows.

Although significant Java expertise is still occasionally needed to extend or maintain this system, most of the integration between Mahout and JRuby doesn't require encyclopedic knowledge of Java and can be accomplished using standard Ruby methods and tools.

17.2 General structure of the email marketing system

The overall data flow at Shop It To Me roughly follows the path shown in figure 17.1. This diagram shows how registration information is used to fill a user table as well as a brand preference table. It also shows how specialized data importer is used to find sale items from retail partners. The core of the system is the SaleMail constructor, which accesses tables containing information about current sale items, users, and their expressed interests. Models, including a click-prediction model, are used to select which items are to be included in each user's email. The items included in each email are recorded in a table of item appearances.

As figure 17.1 shows, users and items for sale form the data that make up the predictors for the click model used by the email generator. When users click on items in the emails that they receive, they cause web page visits to be recorded in log files. These recorded clicks are then turned into training data for the next round of modeling so that the click model can be updated. This updated model is then used for the next round of recommendations, and the cycle repeats. Some of these data sets are stored in relational databases, such as the table of users, but other data sets, such as

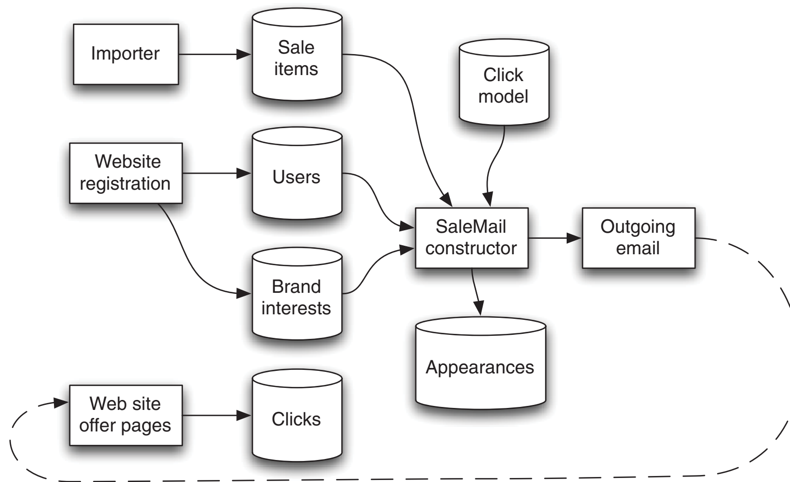


Figure 17.1 The overall data flow for the Shop It To Me email marketing system. The dashed line illustrates how users provide feedback to the system by clicking on items in their emails.

the appearances table, are stored either in large collections of log files or are stored implicitly and must be reconstructed when needed.

The emails generated by the Shop It To Me system look like the one shown in figure 17.2, where you can see the top of an email offering a selection of wristwatches. Many more products are shown further down in the email.

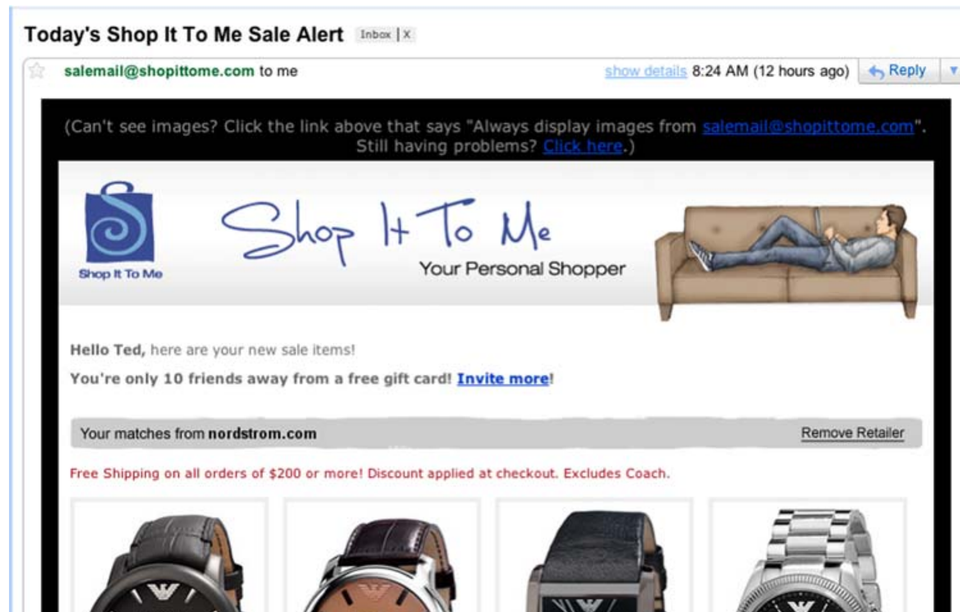


Figure 17.2 Extract from a product suggestion email sent by Shop It To Me

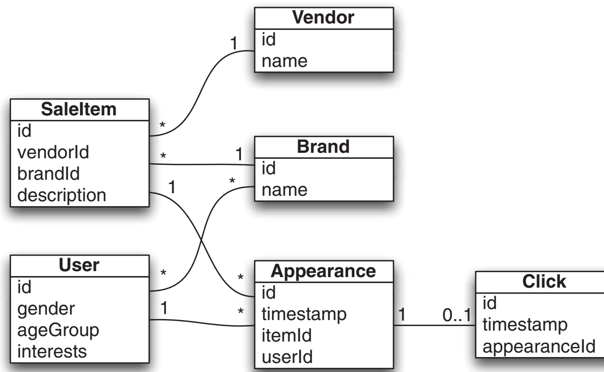


Figure 17.3 Simplified UML view of database tables used to support the data flow at Shop It To Me. Each sale item has a unique vendor and brand but can appear many times. Likewise, a user may indicate interest in multiple brands but will see many items. Each appearance may result in at most one click.

The databases depicted in the workflow of figure 17.1 store the state of the entire system, including the offers that are available, the emails that were sent, and the preferences expressed. Figure 17.3 shows a simplified structure of the tables in the Shop It To Me system that are used to record the required user and item data along with the relevant actions needed to build the click model.

In figure 17.3, denormalized views of the User and SaleItem tables are exported as Apache Hadoop files for use in creating the training data. The Appearance table isn't actually a database table, due to size. Instead it's kept in conventional files that contain logs of emails sent out. The Appearance and Click tables are both exported to Hadoop for joining. Note how these tables are kept in a highly normalized form. This is important for transactional processing and for supporting web access, but this normalization must be removed for training.

You've now seen the framework of the Shop It To Me email system and how their marketing process is structured. Think about how they designed the classification system. Just as with any classifier you might build, the first stage for Shop It To Me was to train a model.

17.3 Training the model

Training a model first requires preliminary planning. You must consider how to pose the questions being asked in order to meet your goals and determine which target variables will be the output of your system. You must examine the available historical data, see what features are available for use as variables, and determine which of these will be most effective as predictor variables. For a Mahout classifier, it's particularly important to scope the proposed project for pain points in order to best determine how to balance speed and size requirements, as you saw in previous chapters.

In this section, you'll see how Shop It To Me accomplished these steps.

17.3.1 Defining the goal of the classification project

The Shop It To Me engineers approached their problem much as we outlined in previous chapters. The goal of their classification project was to produce a model that

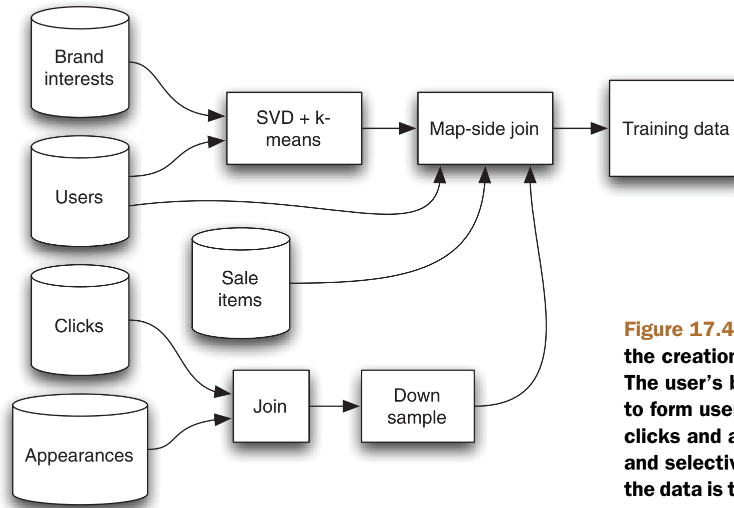


Figure 17.4 The data flow during the creation of model training data. The user's brand interests are used to form user clusters. In parallel, clicks and appearances are joined and selectively downsampled. All of the data is then joined to produce the training data.

could predict whether a user would click on an item using the item's appearance and the user's clicking history as training data. Predicting clicks is the target; user and item characteristics are the predictors. To be usable, however, the Shop It To Me data required significant preprocessing.

The process of creating training data is shown in figure 17.4. This figure shows how user data is preprocessed and joined with item data to form the training data used in model training.

Some of the richest information available in the Shop It To Me system is the brand preference database. When users register with the Shop It To Me system, they're asked which brands they like. Their answers are stored in the brand preference database and are an excellent source of information about what users do or don't like. Unfortunately, this data isn't in a form that's usable for classification.

To rectify this, the data is first reduced using an SVD decomposition and then clustered using the k-means clustering algorithm. The SVD decomposition is a mathematical technique often used to reduce the dimensionality of observed data. This results in the assignment of all users to clusters of people who have expressed similar brand preferences. The brand preference cluster ID expresses the original brand preference information in a form very suitable for modeling.

The Appearances table is the largest table in the entire computation, and each record in the final training data can be traced back to a unique row in this table. Moreover, the rows of the Appearances table are downsampled at different rates depending on whether or not the appearance of the item resulted in a click. Unclicked appearances are downsampled substantially, but clicked appearances aren't downsampled at all. In order to make downstream joins handle less data and thus be more efficient, clicks and appearances are joined in a full reduce join and are downsampled in the reducer before joining them against user and item metadata.

Click and appearance data are then joined against user metadata, including gender, age, general geographic location, and brand preference cluster, as well as against sale item metadata including item identifier, color, size, and description. This second join is done using a map-side join because the user and item metadata tables contain only a few million entries and can be kept in memory relatively easily.

This final join could alternatively have been made into a reduce-side join and integrated into the click and appearance join step. It's possible the performance of this alternative could have been improved with more experimentation.

17.3.2 Partitioning by time

Accumulated training data is partitioned by day to allow flexibility in terms of how much data is used for training and, most importantly, to allow test data to come from disjointed and later times than the training data. This gives a more realistic appraisal of performance because it includes real-world effects like item inventory changes. Daily aggregation also helps when it comes time to expunge old training data that's no longer used in modeling.

Time partitioning also allows training data to be created incrementally, one day at a time. This means that the large cost of building a long-term training set can be paid over a long period of time rather than delaying the build process each day.

17.3.3 Avoiding target leaks

Throughout the chapters on classification, we've emphasized the importance of careful planning for feature extraction to avoid creating target leaks. In order to avoid problems with target leaks, Shop It To Me engineers used a multilayered segregation of data based on time. The SVD and k-means clustering of the brand preference information were based on an early time period and were kept relatively stable as models were rebuilt. The resulting cluster model was applied to data from a later time period to produce the actual training data. Additional, even more recent data was held back from the training algorithm to assess the accuracy of the trained model.

17.3.4 Learning algorithm tweaks

The default behavior of the Mahout learning algorithms was not entirely suited to the task that Shop It To Me needed to do, partly because of the way the classifier was being used as a recommender and partly because the model was difficult to train. These difficulties led the engineers there to extend the default Mahout logistic regression learning algorithm in several ways.

OPTIMIZING BASED ON PER-USER AUC

The `AdaptiveLogisticRegression` learning algorithm uses AUC as a default figure of merit to tune the hyper-parameters for binary models. In the Shop It To Me classification system, AUC is an accurate measure of how well the model predicts which user-item pairs are likely to result in a click. Unfortunately, one of the strongest signals in click-through models involves the classification of users into those who click and those

who don't. Because the fundamental problem in email marketing is to rank items for each user, predicting who will click isn't of much interest. What matters is the prediction of which items the user will click on if they're shown to the user. In order to help the learning algorithm find models that solve the right problem, a different figure of merit than normal was required.

In Mahout, there are two classes that can be used to compute different forms of AUC. Normal global AUC is computed by the `GlobalOnlineAuc` class, whereas AUC based on rankings within groups is computed using `GroupedOnlineAuc`. The `AdaptiveLogisticRegression` learning algorithm allows alternative AUC implementations and a grouping criterion to be supplied. Grouping on user or user cluster allows models that are good recommenders to be distinguished from models that simply pick users who click.

MIXED RANK AND SCORE LEARNING

Whether learning a probability directly is better than learning an ordering of results is a major current area of research. These approaches seem equivalent at first examination, but several researchers have noted that there are differences in practical situations. One recent advance is represented by the paper "Combined Regression and Ranking" by Google researcher D. Sculley.[1] Shop It To Me engineers implemented this technique in a wrapper around Mahout code by maintaining a short history of training examples and using either the current training example or the difference between the current example and a previous one to compute the gradient for updating the model. Mahout has a largely untested implementation of this technique in the `MixedGradient` class.

This technique was quite effective for Shop It To Me, but more experience is necessary to determine how often this will be true.

17.3.5 *Feature vector encoding*

Figure 17.5 shows a record from an idealized set of training data that might be used to build a click model. In this model, as in more realistic versions of such a record, there are three types of fields that are of interest:

- User-specific fields
- Item-specific fields
- User-item interaction fields

The user-specific fields include values for variables such as the age group, gender, and brand preference cluster. The item-specific fields include values for variables such as the vendor, brand, and item description. Of greatest benefit are the user-item interaction variables. The interaction variables enable the model to be more than a record of what items or kinds of items are most popular.

---

[1] D. Sculley, "Combined Regression and Ranking" http://www.eecs.tufts.edu/~dsculley/papers/combined-ranking-and-regression.pdf. See also the video lecture: http://videolectures.net/kdd2010_sculley_crr/



Figure 17.5 Fields in a possible training record for a simplified email marketing, click-through prediction model. The record contains user-specific, item-specific, and user-item interaction variables. Real models of this sort have many more fields than are shown in this conceptual example.

The interesting item here in contrast with our previous examples of feature encoding is the interaction variable that involves the user's brand preference cluster identifier (*cluster* in the diagram), gender, and item brand.

To encode an interaction variable using the feature-hashing style recommended for SGD learning, the code needs to pick feature vector locations are that are independent of the feature vector locations used for each of the constituent features in the interaction. For example, figure 17.6 shows how we'd like things to work. Assuming we have a sparse feature vector of size 100, the locations for the feature "gender = female" might be assigned locations 8 and 21 by the hashed feature encoding. Similarly, "item = dress" might be assigned locations 77 and 87 in the vector. The interaction "female x dress" needs to be assigned locations that are as statistically independent from these four locations as possible, while still being deterministically computed based on these original values.

One way that Shop It To Me engineers were able to get statistical independence for the locations was to use random seeds to combine the locations of each of the two locations. The seeds generated by the random number generator for this example were 22 and 92. To get the i-th location for the interaction feature, use

```
hash[i] = seed[0] * x[i] + seed[1] * y[i]
```

where x and y contain the feature locations for "woman" and "dress" respectively.

This method is closely related to double-hashing as used in Bloom filters, and it can be shown to provide reasonably good, independent locations for the composite features. The following listing provides a sample encoder with one implementation of an interaction encoder that's based on this idea.

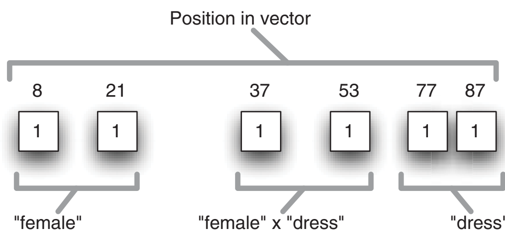


Figure 17.6 Interaction features need to be hashed to locations different from the locations of their component features.

Listing 17.1 Sample encoder showing one implementation of an interaction encoder

```
public class CategoryInteractionEncoder {
  private int[] seeds;
  private CategoryFeatureEncoder[] encoders;
  private int probes = 2;
  CategoryInteractionEncoder(int seed, CategoryFeatureEncoder... enc) {
    Random r = new Random(seed);
    seeds = new int[enc.length];
    for (int i = 0; i < enc.length; i++) {
      seeds[i] = r.nextInt();
    }
    this.encoders = enc;
  }
  public void addToVector(int[] categories, double weight, Vector data) {
    int[] hashes = new int[categories.length];
    for (int i = 0; i < probes; i++) {
      for (int j = 0; j < categories.length; j++) {
        hashes[i] += seeds[j] *
           encoders[j].hashForProbe(categories[j], i);
      }
    }
    int n = data.size();
    for (int h : hashes) {
      h = h % n;
      if (h < 0) {
        h += n;
      }
      data.setQuick(h, data.getQuick(h) + weight);
    }
  }
  ...
}
```

❶ Seeds combine hashes
❷ Encoders encode constituent variables
❸ Initialize seeds deterministically
Combine hashes
❹ Combine hashes of each feature
Use combined hashes to set values

The seeds ❶ and encoders ❷ are kept around in the interaction encoder. The seeds are initialized ❸ in the constructor, which is where the array of encoders is passed in. In the `addToVector()` method, hashes from the encoders for each variable in the interaction are combined with integer multipliers ❹ that were randomly chosen when the interaction encoder was constructed, based on the seed supplied at that time. This multiplicative combination ensures that none of the feature locations chosen for the interaction encoder are likely to collide with any of the locations chosen for the constituent variables. Note that `CategoryFeatureEncoder` is specific to this example and is not part of Mahout itself.

There are other possible approaches to interaction encoding. For instance, two sums of the locations of the first and second hashed locations for all constituent encoders could be kept and then combined using the standard double-hashing algorithm. The following code shows how this alternative encoding approach could work:

```
public class CategoryInteractionDoubleHashingEncoder {
  private CategoryFeatureEncoder[] encoders;
  private int probes = 2;
  CategoryInteractionEncoder(CategoryFeatureEncoder... enc) {
    this.encoders = enc;
```

❶ Encode constituent variables

```
  }
  public void addToVector(int[] categories, double weight, Vector data) {
    int h0 = 0, h1 = 0;
    for (int j = 0; j < categories.length; j++) {
      h0 += encoders[j].hashForProbe(categories[j], 0);
      h1 += encoders[j].hashForProbe(categories[j], 1);
    }
    int n = data.size();
    for (int i = 0; i < probes; i++) {
      h = h0 + i * h1;
      if (h < 0) {
        h += n;
      }
      data.setQuick(h, data.getQuick(h) + weight);
    }
  }
  ...
}
```

❷ Compute two base hashes

❸ Use combined hashes to set values

Here you have encoders ❶ for each constituent variable, just as before, but you only use the sums ❷ of the first and second probes for each constituent variable. When computing the locations ❸ for the interaction, combine these two sums using double hashing.

Neither of these approaches has been rigorously evaluated, but both seem to have worked well in practice. Ironically, to use a classifier for recommendations, interaction variables between user and item are required, but having these interactions increases the work involved in encoding variables and computing classification results. Because the Shop It To Me classification throughput requirement is so high, that's a big deal and has to be addressed by speedups in the classification system. These speedups avoid both the encoding cost and the computation cost of the classification computation itself.

17.4 Speeding up classification

As mentioned before, Shop It To Me had a firm requirement for performing classifications at a prodigious rate in their final system. In order to perform the required tens of millions of classifications per second on the available few machines, several improvements were needed over and above what base Mahout provides. Because the internal feature vector used has a length of over 100,000 elements, meeting this classification rate in a naive way would require a raw numerical throughput of several trillion floating-point operations per second, even if you don't count feature encoding. With only a few dozen CPU cores available for the task, this wasn't plausible.

To meet these requirements, Shop It To Me engineers made improvements that included the following:

- Caching partially encoded feature vectors
- Caching hash locations within encoders
- Splitting the computation of the model into pieces that can be separately cached

These improvements ultimately reduced the runtime model computation to three lookups and two additions. The necessary precomputation involved one encoding and model computation per user and a few hundred such computations per item. The overall savings due to these changes were about 3–4 orders of magnitude, which makes the overall model computation quite feasible. The improvements listed here may look relatively simple, but they provide a powerful extension of the way Mahout can be used for classification, and you may want to consider trying innovations such as these in your projects if your required throughput is as high as Shop It To Me's is.

17.4.1 Linear combination of feature vectors

The feature-extraction system used in Mahout works by adding the vectors corresponding to separate features in order to get an overall feature vector. In the case of Shop It To Me, the feature vector can be divided into three parts related to the user, the item part, and both the user and item.

$$x = x_{user} + x_{(cluster + ageGroup)\ \mathrm{x}\ itemBrand} + x_{item}$$

This division into parts can clearly help with feature-encoding costs because you can cache these components instead of computing them on the fly each time you want to evaluate the model.

Caching the entire feature vector doesn't work well because of the very large number of values that would have to be cached. Caching the pieces works because the number of unique pieces is much smaller. Essentially, the larger cache has a required size that's nearly the product of the sizes of the smaller caches.

Figure 17.7 shows how the three parts of the feature extraction are used in the computation of the model score.

Using caching for these three components helped with speed somewhat, but, by itself, this caching did not improve speed enough. It did, however, lay the groundwork for additional improvements.

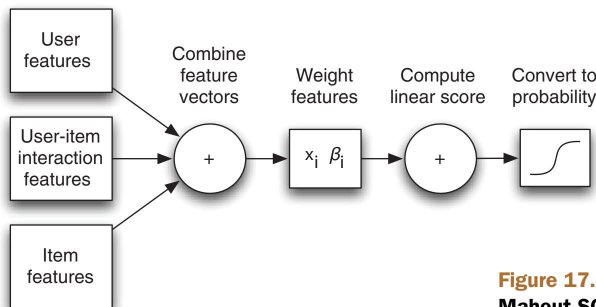


Figure 17.7 How model scoring works for the Mahout SGD logistic regression model

17.4.2 Linear expansion of model score

Splitting the feature vector into three pieces is useful for speeding up feature encoding, but the model computation that uses the feature vector is composed of linear operations all the way to the last step. This allows even more aggressive caching to be done. Moreover, the items cached at the later point in the computation are much smaller.

The way that logistic regression models such as the SGD models in Mahout work is that they produce a weighted sum of the features and then transform that weighted sum so that it's always in the range from 0 to 1. When ranking values, you can omit that last transformation without disturbing the ordering of the final results. In Mahout, the `classifyScalarNoLink(Vector)` method in `AbstractVectorClassifier` omits this final transformation exactly as desired.

Once this final transformation is removed and the model is simplified to a weighted sum, you can use the linear combination property of feature vectors. Using the dot product notation, the model output, z, can be written this way:

$$z = \beta \cdot x$$

Here β represents the weights the model applies to each feature, and x represents the features that are given to the model. Both quantities are vector-valued. The multiplication here is a dot product that indicates the sum of the products of corresponding elements of ? and x. The dot product is where all of the intensive computation happens. Note that this formula can be broken into three parts, just as the feature vector can be split:

$$z = \beta \cdot x_{user} + \beta \cdot x_{(cluster + ageGroup) \text{ x } itemBrand} + \beta \cdot x_{item}$$

Pictorially, this split can be depicted by changing figure 17.7, which showed how the model computation works in Mahout's SGD models, so that more computations are done separately before being finally combined.

Figure 17.8 shows user, item, and user-item features being applied separately to the weight vector, β, before the three components of the score are combined. The clever thing about this is that these three parts can now be efficiently cached.

Importantly, the value of each part is a single floating-point number rather than a vector, so in the case of a full cache hit, the cost of a single model computation for a

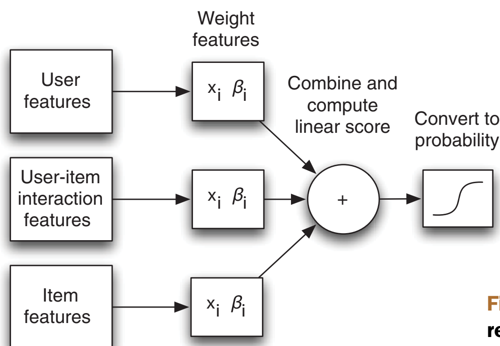


Figure 17.8 How model scoring can be rearranged to allow precomputation and caching of intermediate results

particular user and item computation drops from several thousand floating-point operations to just two table lookups and two additions. There are only two lookups because the user component of the score is computed outside the inner loop. Caching smaller objects also decreases memory consumption, allowing more items to be cached.

The following listing shows how this works in code.

Listing 17.2 Using caching and partial model evaluation to speed up Item selection

```
public class ModelEvaluator {
  private OnlineLogisticRegression model;
  private List<Item> items = Lists.newArrayList();
  private Map<Item, Double> itemCache = Maps.newHashMap();
  private Map<Long, Double> interactionCache = Maps.newHashMap();
  private FeatureEncoder encoder = new FeatureEncoder();
  public List<ScoredItem> topItems(User u, int limit) {
    Vector userVector =
      new RandomAccessSparseVector(model.numFeatures());
    encoder.addUserFeatures(u, userVector);
    double userScore = model.classifyScalarNoLink(userVector);
    PriorityQueue<ScoredItem> r =
            new PriorityQueue<ScoredItem>();
    for (Item item : items) {
      Double itemScore = itemCache.get(item);
      if (itemScore == null) {
        Vector v = new RandomAccessSparseVector(model.numFeatures());
        encoder.addItemFeatures(item, v);
        itemScore = model.classifyScalarNoLink(v);
        itemCache.put(item, itemScore);
      }
      long code = encoder.interactionHash(u, item);
      Double interactionScore = interactionCache.get(code);
      if (interactionScore == null) {
        Vector v = new RandomAccessSparseVector(model.numFeatures());
        encoder.addInteractions(u, item, v);
        interactionScore = model.classifyScalarNoLink(v);
        interactionCache.put(code, interactionScore);
      }
      double score = userScore + itemScore
                    + interactionScore;
      r.add(new ScoredItem(score, item));
      while (r.size() > limit) {
        r.poll();
      }
    }
    return Lists.newArrayList(r);
  }
}
```

1. Receive from other code
2. Score user only once
3. Accumulate best priority queue
- Look for cached item score
4. Compute score if not found
5. Look for cached interaction effect
6. Combine components

This class shows some simplified code that does the model evaluation using caches for item and interaction effects. Assume in this code that the model and the list of items will be populated by some other means 1.

What this class does when given a user is to first compute the score component for that user ❷. Then it scans the list of all items and computes the model score for each item. The items with the top score are retained ❸. The score is computed in three parts. The part due to the user variables is computed outside the item loop ❷ and retained. Then inside the inner loop, the item ❹ and user-item interaction ❺ parts are each computed if they aren't found in the appropriate cache data structure. The `FeatureEncoder` class is a convenience class that's used to wrap the feature encoders for the various user and item features and their interactions. Finally the components of the score are combined to get the final score ❻.

Clearly it isn't as simple to split up the model computation in this way, but in some cases it's worthwhile due to the massive speedups available.

17.5 Summary

The online marketing case study presented in this chapter describes the innovative approaches used by Shop It To Me engineers in developing a Mahout classifier for predicting clicks for on-sale items presented to customers. The scale and the speed that Shop It To Me was able to achieve using Mahout classifiers both in training and when running a deployed model against live data illustrate how Mahout can be used in very large-scale problems.

The principles you have learned throughout the chapters on classification were applied in this real-world classifier. One important concept is the idea that classification can be a useful way to achieve results similar to recommendation in systems that preclude traditional recommendation approaches.

Another important principle that's illustrated in this case study is the value of scoping out the problem carefully at the start, looking particularly for so-called pain points that could be a barrier to success if not addressed by the engineering design. These pain points are often caused by extreme requirements for speed and volume, and Shop It To Me's online marketing system illustrated those challenges in abundance. Scoping the problem led engineers to recognize the value of switching to Mahout classification for this system and identified the need to extend the effectiveness of Mahout by some customization.

The Shop It To Me example also shows the power of using large-scale joins to implement the training pipeline in parallel, a take-home lesson described in the previous chapter. You will likely find many applications where this way of building a pipeline helps you meet the requirements for scale and speed of extremely large projects.

Other key ideas that this case study demonstrates are the innovations in the Shop It To Me system. These innovations include the use of ranking objectives for learning models and the use of grouped AUC for meta-learning. Learning with a composite goal that involves ranking and regression together appears to be a very powerful technique for nearly all applications of SGD learning. Grouped AUC as a target for the `AdaptiveLogisticRegression` to optimize learning meta-parameters looks like a very effective strategy whenever classifiers are used as recommenders.

In terms of throughput, the big lesson is the way that caching strategies were used to speed up the Mahout classifier. These methods included caching of partially encoded feature vectors, caching of hash locations within encoders, and splitting the computation of the model into pieces that can be separately cached. Even with the innovations in the Shop It To Me approach to using Mahout for classification, this isn't the end of the line by any means. There are considerable degrees of improvement in training speed still available, and it's unlikely that the last ounce of speed has been milked from the deployed models.

Just how far Mahout can go is a subject for the next chapter. And that's the chapter you get to write.

appendix A
JVM tuning

This appendix discusses ways to increase the performance of nondistributed Mahout-based applications by tuning the settings of the JVM. It's *not* a guide to tuning Hadoop, which is a large topic in its own right. This tuning is relevant to nondistributed aspects of Mahout, which are almost entirely the recommender engine implementations. While the settings here are likely beneficial to any Java-based server process, the advice is specifically directed to those running nondistributed Mahout-based recommender engines.

When using a data set of substantial size—perhaps ten million preferences and up—it'll be well worth tuning the JVM settings for performance. It's mostly the heap-related (memory-related) settings that are important. Optimal settings depend on the operating system, available resources, architecture, and JVM, but the following JVM flag settings will establish a good starting point.

The flags listed in table A.1 are valid on all known Java 6 JVMs, which Mahout requires, with the exception of `-XX:NewRatio`, as explained in the table.

Table A.1 Key JVM tuning parameters for recommender engines

Parameter	Description
`-Xmx`	This selects the maximum size that Java will allow the heap to grow to. For example, `-Xmx512m` indicates that the heap may grow to 512 MB.
`-server`	Modern JVMs have two important flags, `-client` and `-server`, which select JVM settings broadly appropriate for short-running, small-footprint, client-side applications, and long-running, resource-intensive, server applications. The default depends on the JVM and environment. Obviously, `-server` is appropriate here.
`-d32` and `-d64`	These request 32-bit and 64-bit mode, respectively. On a 64-bit machine, both are possible. While it's typically best to let the JVM decide, selecting 32-bit mode will decrease memory requirements. Of course, 32-bit mode makes it impossible to use more than 2–3 GB of heap (depends on the JVM), but, when pushing this boundary, a small memory savings may be worthwhile. Selecting 32-bit mode on a 64-bit machine may incur a small performance penalty.

Table A.1 Key JVM tuning parameters for recommender engines *(continued)*

Parameter	Description
`-XX:+NewRatio=`	Some heap space is always reserved for short-lived, temporary objects, and can never be used by more permanent data structures. Mahout runs lean: it creates few short-lived objects but consumes a lot of heap space with long-lived objects. The default proportion of the heap reserved for short-lived objects is generally too large and ends up being wasted needlessly. A setting of 12, for example, requests that only 1/12 of the heap be reserved for short-lived objects. Note that this setting appears to be specific to Sun JVMs only.
`-XX:+UseParallelGC` and `-XX:+UseParallelOldGC`	This instructs the JVM to make better use of multiple processors or cores (common on even desktop computers today) by running garbage collection in parallel.

To demonstrate the effect of these JVM settings, we experiment with the code from listing 4.2 (in chapter 4). Let's start with 32-bit client JVM defaults: `-client -d32 -Xmx512m`. These are JVM flags, not program flags, so they go before the program class name. On one modern 64-bit computer we used for testing, the load evaluation result indicates that it's producing recommendations in about 425 ms and consuming 248 MB of heap in steady state. (Your timing results may vary, as may your exact heap size.)

What if we substitute `-server` for `-client`? If anything, it should increase performance. Memory requirements stay the same, but recommendation time drops to 192 ms. This shows how much more appropriate the server JVM tuning is for an application like this.

Now, change `-d32` to `-d64`. You'll encounter an `OutOfMemoryError`. You'll probably need to allocate 768 MB to the heap, with `-Xmx768m`, to get it to run again. Recommendation time drops again to 142 ms. Not surprisingly, 64-bit mode is more efficient on a 64-bit machine. Steady-state memory requirements are about the same though: 256 MB. The 64-bit mode increases the memory consumed by objects and references, but by design, relatively few long-lived objects are created by Mahout's recommender engine to begin with.

You may have noticed a discrepancy: why, if steady-state memory requirements are only about 256 MB, did the JVM fail due to lack of heap space when it had 512 MB available? Memory requirements peak at a higher level during construction of the in-memory data representation in the `DataModel`.

You should find that the `-XX:+NewRatio=12` flag allows you to reduce heap size to about 640 MB.

Finally, try adding `-XX:+UseParallelGC -XX:+UseParallelOldGC`. Assuming more than one processor core is available, this will allow garbage collection work to proceed in parallel with the main computation. On our test machine, recommendation time drops to 126 ms. Compare that to the initial result of 425 ms.

Figure A.1 shows how, on a modern machine, improper JVM configuration can slow down performance by about 350 percent! Any serious deployment of a recommender engine must be properly tuned.

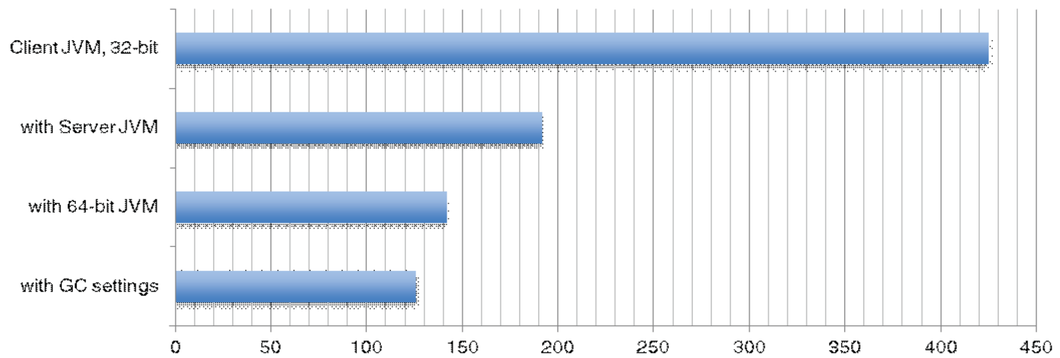


Figure A.1 Progressively better per-recommendation running times (in milliseconds) with various JVM tuning settings

appendix B Mahout math

Many algorithms implemented by Mahout rely heavily on vector and matrix math. Mahout has its own self-contained vector and matrix math library in the math module that can be reused separately from Mahout itself. It is actually an adapted form of CERN's Colt library (http://dsd.lbl.gov/~hoschek/colt/).

This appendix provides a practical overview of key parts of this math module that users encounter frequently while using Mahout: `Vector` and `Matrix` instances, and their associated operations. It is not exhaustive documentation; the Mahout project Javadoc and source code can provide more detail if you're interested.

B.1 Vectors

Vector means slightly different things in different contexts, though all of them are similar. Most people encounter a vector first in physics, where it denotes a direction and is typically illustrated by an arrow. For our purposes, we sometimes use vectors to represent *points* instead. This isn't such a different idea; a point merely corresponds to a vector from the origin to that point.

Most often in machine learning and Mahout, we use vectors in a more abstract sense, one that doesn't necessarily map to a geometric interpretation. A vector can be merely a tuple, an ordered list of numbers. It has a size (or cardinality), and at each index (position) from 0 to size – 1 the vector has some numeric value. Vectors are usually written as a list of numbers like so: (2.3, 1.55, 0.0). This is a vector of size 3, whose value at index 1 is 1.55, for example. In machine learning, we sometimes deal with vectors with sizes in the millions.

B.1.1 Vector implementation

`Vector` in `org.apache.mahout.math` is the interface that all implementations implement. There are several implementations of this interface. Representing vectors in a memory-efficient way, and accessing their values efficiently, are quite important to Mahout, and the best way to do these things varies according to the nature of the vector and how it'll be used.

The *sparseness* or *denseness* of a vector's data is the most important consideration. In many cases, a vector has a (nonzero) value at only a few indices relative to its size. The value at all other indices is implicitly zero. If we had a vector of size 1,000 and only 2

indices had a nonzero value, it wouldn't make much sense to store 998 zeroes in memory along with those 2 nonzero values. But it wouldn't seem so wasteful if only 100 values were zero.

A vector with relatively many nonzero values in relation to its size is called dense. Such a vector can be represented with an implementation that stores values in an array of `doubles`. Vector indices correspond directly to array indices. A dense vector's advantage is speed: being array-backed, it's quick to access and update any of its values. `DenseVector` provides such an implementation.

Sparse vectors call for a different implementation. In order to not consume memory storing values for zero entries, they instead only store values and indices with nonzero values. A hash table could be used to implement a simple index-to-value mapping. Accessing an index value is slower than with direct array access, but not by much. `RandomAccessSparseVector` implements this idea.

The problem with a hash-backed implementation is that it becomes relatively slow to iterate through all values in order by index, and this is frequently required. An ordered mapping based on a tree structure or something similar can address this problem, because it maintains keys in order. The price of this feature is longer access time. `SequentialAccessSparseVector` is the third and final implementation, providing a `Vector` with these characteristics. Figure B.1 shows an example of dense and sparse vectors.

Dense	3.5	0.0	0.0	-1.2	2.0	0.0	-3.3
	0	1	2	3	4	5	6
Sparse	3.5	0.0	0.0	-1.2	2.0	0.0	-3.3
	0	1	2	3	4	5	6

Figure B.1 A dense vector is a represented as a full array, so it needs to store only the double values. A sparse vector doesn't allocate the space needed for zero-valued cells, so it keeps an integer to indicate the index of every nonzero double value. Stored values are boxed; others are implicit and aren't stored explicitly.

B.1.2 Vector operations

Now we'll examine the frequently-used methods of `Vector`.

The `get(int)` and `set(int, double)` methods provide the basic operations to access and change vector values at an index. The `getQuick(int)` and `setQuick(int, double)` methods do the same, but without bounds checking, which becomes a useful performance optimization internally. Other callers should likely stick to the first two methods. The `size()` method, as expected, returns the vector's size.

Iterating over all elements is accomplished through `iterator()` and `iterateNonZero()`, which iterate over all indices and values, or only those with a nonzero value. The `Iterator` returned from these methods provides `Element` objects, encapsulating an index and value.

The standard `clone()` method is implemented here to copy a `Vector`; implementations provide copy constructors as well. The `like()` method returns an empty `Vector` of the same class. This is a form of factory method.

Finally, we can talk about where the mathematics comes into this math module. The `plus(Vector)` and `minus(Vector)` methods provide vector standard addition and subtraction. You can multiply or divide all values by a scalar with `times(double)` or `divide(double)`. Also note methods like `dot(Vector)`, which computes a dot product (inner product), and `cross(Vector)`, which creates a matrix from the pair-wise value products of two vectors.

B.1.3 Advanced Vector methods

So far, all of this should be straightforward and as expected. The first non-trivial method in `Vector` is `assign(Vector, DoubleDoubleFunction)`. It mutates the `Vector` and sets its value equal to the result of some function of the values in the `Vector` and another `Vector`. `DoubleDoubleFunction` encapsulates a function of two numeric values that results in another numeric value. This method doesn't let you do things you couldn't otherwise accomplish with the previous basic operations, but it may let you do them more efficiently.

For example, given vectors A and B, consider computing $C = mA - B$, where m is a scalar value. This could be implemented as in `myOperation`.

Listing B.1 Computing `Vector` operations efficiently

```
Vector myOperation(Vector A, Vector B, double m) {
  return A.times(m).minus(B);
}

Vector myFasterOperation(Vector A, Vector B, final double m) {
  A.assign(B, new DoubleDoubleFunction() {
    public double apply(double a, double b) {
      return a * m - b;
    }
  });
}
```

Both `myOperation` and `myFasterOperation` accomplish the task. The first is simple, but slower than it appears. Each of the two mathematical operations allocates an entirely new `Vector`, and the `Vector`s are traversed twice. The second method, however, doesn't allocate a new `Vector`—it destructively changes *A*, which may be entirely fine and desirable. It also needs only one pass over the vectors to compute the result.

The `aggregate(DoubleDoubleFunction, DoubleFunction)` method can simplify computing a function of all values in a vector, especially when used in conjunction with ready-made functions in `Functions`. For instance, to compute the sum of squares of values in a `Vector`, you could make use of either method in the following listing.

Listing B.2 Computing aggregations efficiently

```
double mySumOfSquares(Vector A) {
  double sum = 0.0;
  for (Element e : A.iterateNonZero()) {
    sum += e.get() * e.get();
  }
  return sum;
}

double myOtherSumOfSquares(Vector A) {
  return A.aggregate(Functions.PLUS, Functions.SQUARE);
}
```

The second method is not, in this case, more efficient, but it is more compact.

B.2 Matrices

Matrices, unlike vectors, are unambiguous and likely familiar—you'll likely have at least glimpsed matrix operations in a mathematics course. A matrix is a table of numbers. This simple construct has proved tremendously useful for representing concepts in physics and linear algebra—and by extension machine learning. Operations on matrices are a fundamental part of algorithms in these fields, and Mahout uses matrices in many places. Often, in Mahout, it'll be useful to think of the rows or columns of a matrix as vectors in their own right.

A matrix is represented by an implementation of the `Matrix` interface in Mahout. Its implementation follows the same pattern as `Vector`: several different classes implement `Matrix` in order to provide performance characteristics optimized for different situations and usage patterns. The same sparseness consideration similarly applies to matrices. `SparseMatrix` and `DenseMatrix` represent matrices with few (nonzero) values, and with many values relative to their size, respectively.

`SparseRowMatrix` and `SparseColumnMatrix` are interesting variants on `SparseMatrix` that are intended for use where rows (or columns) of a matrix will frequently be accessed as a unit, as a vector. `SparseRowMatrix` supports use cases where a matrix has relatively few nonzero rows, but rows must be accessed as row vectors. `SparseColumnMatrix` behaves similarly for columns.

B.2.1 Matrix operations

Many of the methods found in `Vector` also exist in `Matrix`, in a slightly different form, to accommodate the matrix's two-dimensional nature. The `get(int, int)` method returns the value at a given row and column, for example. The `size()` method returns both the row and column size of the matrix. The `like()` and `clone()` methods are implemented as well. Label bindings for rows and columns are supported, as are convenience operations like `assign(double)`.

`Matrix` implementations provide mathematical operations like `plus(Matrix)` and `times(Matrix)` to implement matrix addition and multiplication. Other common

matrix-specific operations, like `transpose()` to generate the transposed matrix, and `determinant()` to compute the determinant, exist as well.

Matrix operations are largely unsurprising, but they make it easy to implement otherwise complex operations in a compact way. For example, the following listing illustrates multiplying an m x n matrix against a vector of size n (which can be thought of as the lone column vector of an n x 1 matrix).

Listing B.3 Multiplying a matrix and a vector

```
Vector vector = new SequentialAccessSparseVector(n);
Matrix matrix = new SparseMatrix(new int[] {m, n});
Matrix vectorAsMatrix = new SparseColumnMatrix(new int[] {n, 1});
vectorAsMatrix.assignColumn(0, vector);
Matrix productMatrix = matrix.times(vectorAsMatrix);
Vector product = productMatrix.getColumn(0);
```

This sort of operation occurs frequently in linear algebra and machine learning, and you can see how relatively simple it can be to implement it with Mahout's math module.

B.3 Mahout math and Hadoop

`Matrix` and `Vector` implementations are frequently used in Hadoop-related mapper and reducer jobs. This means they must necessarily be serializable—convertible into a sequence of bytes that can be stored, transmitted, and reconstituted. Hadoop doesn't employ Java's standard serialization mechanism via `java.io.Serializable` to accomplish this. Instead, it defines a very similar interface called `Writable` for this purpose. Classes implement `Writable` in order to make themselves usable as keys and values in Hadoop.

`Vector` and `Matrix` do not themselves extend `Writable`, because this would make the math module depend on Hadoop, and conceptually it functions independently of something like Hadoop. Within the core Mahout module, you will find instead `VectorWritable` and `MatrixWritable`. These implement both `Writable` and `Vector` or `Matrix`, respectively. They encapsulate knowledge of how to serialize a vector and matrix within Hadoop. In `Mapper` and `Reducer` jobs that read or write vectors or matrices, raw implementations will be wrapped in such classes before passing them to Hadoop, and they'll be returned in this form.

appendix C Resources

The internet can provide more, and more recent, resources related to machine learning and Mahout than we can here. Search for *collaborative filtering, clustering, classification,* and other such topics in your favorite search engine. Or try research-specific search engines like Google Scholar (http://scholar.google.com).

In this appendix, we present classic papers and references that will be of interest to anyone looking to better understand the underpinnings of the content of Mahout and this book.

Sources

Anderson, M., M. Ball, H. Boley, S. Greene, N. Howse, D. Lemire, and S. McGrath. "RACOFI: A Rule-Applying Collaborative Filtering System." *Proceedings of COLA '03* (2003). www.daniel-lemire.com/fr/documents/publications/racofi_nrc.pdf.

Blei, D.M., A.Y. Ng, and M.I. Jordan. "Latent Dirichlet Allocation." *Journal of Machine Learning Research* 3 (2003): 993–1022. www.cs.princeton.edu/~blei/papers/BleiNgJordan2003.pdf.

Breese, J.S., D. Heckerman, and C. Kadie. "Empirical Analysis of Predictive Algorithms for Collaborative Filtering." *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence* (1998). http://research.microsoft.com/research/pubs/view.aspx?tr_id=166.

Dunning, T. "Recorded Step Directional Mutation for Faster Convergence." *Computing Research Repository* (2008): http://arxiv.org/abs/0803.3838.

Gath, I., and A.B. Gev. "Unsupervised Optimal Fuzzy Clustering." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11 (1989): 773–780. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.2648&rep=rep1&type=pdf.

Herlocker, J.L., J.A. Konstan, A. Borchers, and J. Riedl. "An Algorithmic Framework for Performing Collaborative Filtering." *Proceedings of the 22nd annual international ACM SIGIR Conference on Research and Development in Information Retrieval* (1999): 230–237. www.grouplens.org/papers/pdf/algs.pdf.

Kannan, R., S. Vempala, and A. Vetta. "On Clusterings: Good, Bad and Spectral." *Journal of the ACM* 51 (2004): 497–515. doi: 10.1145/990308.990313.

Lemire, D., and A. Maclachlan. "Slope One Predictors for Online Rating-Based Collaborative Filtering." *Proceedings of SIAM Data Mining* (2005). www.daniel-lemire.com/fr/documents/publications/lemiremaclachlan_sdm05.pdf.

Lloyd, S.P. "Least Squares Quantization in PCM." *IEEE Transactions on Information Theory* IT-28 (1982): 129–37. http://www.cs.nyu.edu/~roweis/csc2515-2006/readings/lloyd57.pdf.

Lyon, C. "Movie Recommender." CSCI E-280 Final Project, Harvard University, 2004. http://materialobjects.com/cf/MovieRecommender.pdf.

McCallum, A., K. Nigam, and L.H. Ungar. "Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching." *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2000): 169–178. www.kamalnigam.com/papers/canopy-kdd00.pdf

Murtagh, F. "A Survey of Recent Advances in Hierarchical Clustering Algorithms" *The Computer Journal* (1983): 345–359. http://comjnl.oxfordjournals.org/content/26/4/354.full.pdf.

Ng, R.T., and J. Han. "Efficient and Effective Clustering Methods for Spatial Data Mining." *Proceedings of the 20th International Conference on Very Large Data Bases* (1994): 144–155. www.cis.temple.edu/~vasilis/Courses/CIS750/Papers/ClusteringSpatial-Ng.pdf.

Olson, C.F. "Parallel Algorithms for Hierarchical Clustering." University of California at Berkeley, Berkeley, CA, 1994. www.eecs.berkeley.edu/Pubs/TechRpts/1994/CSD-94-786.pdf.

Rajaraman, A., and J.D. Ullman. "Mining of Massive Data Sets," 2010. http://infolab.stanford.edu/~ullman/mmds.html.

Rand, W.M. "Objective criteria for the evaluation of clustering methods." *Journal of the American Statistical Association* (1971): 846–850. doi: 10.2307/2284239.

Rennie, J., Shih, L., Teevan, J., and Karger, D. "Tackling the Poor Assumptions of Naive Bayes Text Classifiers." *In Proceedings of the Twentieth International Conference on Machine Learning* (2003):616-623. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.322.

Resnick, P., N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. "GroupLens: An Open Architecture for Collaborative Filtering of Netnews." *Proceedings of the 1994 ACM conference on Computer Supported Cooperative Work* (1994): 175–186, http://portal.acm.org/citation.cfm?id=192905.

Sarwar, B., G. Karypis, J. Konstan, and J. Riedl. "Item-Based Collaborative Filtering Recommendation Algorithms." *Proceedings of the Tenth International Conference on the World Wide Web* (2001): 285–295. http://portal.acm.org/citation.cfm?id=372071.

Sculley, D. "Combined Regression and Ranking." *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2010). www.eecs.tufts.edu/~dsculley/papers/combined-ranking-and-regression.pdf. See also the video lecture: http://videolectures.net/kdd2010_sculley_crr/.

index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

JAVA/MACHINE LEARNING

Mahout IN ACTION

Owen • Anil • Dunning • Friedman



A computer system that learns and adapts as it collects data can be really powerful. Mahout, Apache's open source machine learning project, captures the core algorithms of recommendation systems, classification, and clustering in ready-to-use, scalable libraries. With Mahout, you can immediately apply to your own projects the machine learning techniques that drive Amazon, Netflix, and others.

This book covers machine learning using Apache Mahout. Based on experience with real-world applications, it introduces practical use cases and illustrates how Mahout can be applied to solve them. It places particular focus on issues of scalability and how to apply these techniques against large data sets using the Apache Hadoop framework.

What's Inside

- Use group data to make individual recommendations
- Find logical clusters within your data
- Filter and refine with on-the-fly classification
- Free audio and video extras

This book is written for developers familiar with Java—no prior experience with Mahout is assumed.

Sean Owen helped build Google's Mobile Web search and launched the Taste framework, now part of Mahout. **Robin Anil** contributed the Bayes classifier and frequent pattern mining implementations to Mahout. **Ted Dunning** contributed to the Mahout clustering, classification, and matrix decomposition algorithms. **Ellen Friedman** is an experienced writer with a doctorate in biochemistry.

For access to the book's forum and a free audio- and video-enhanced ebook for owners of this book, go to manning.com/MahoutinAction

"A hands-on discussion of machine learning with Mahout."

—Isabel Drost
Cofounder Apache Mahout

"The writing makes a complex topic easy to understand."

—Rick Wagner, Red Hat

"Essential Mahout, authored by the core developer team."

—Philipp K. Janert
author of *Gnuplot in Action*

"Dramatically reduces the learning curve."

—David Grossman
Illinois Institute of Technology

"Recommendations, clustering, and classification all lucidly explained."

—John S. Griffin, Overstock.com

ISBN-13: 978-1-935182-68-9
ISBN-10: 1-935182-68-4



MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]